AFIT/GE/ENG/92D-08

ETANN HARDWARE IMPLEMENTATION FOR RADAR EMITTER IDENTIFICATION

THESIS

James B. Calvin, Jr.
Captain, USAF

AFIT/GE/ENG/92D-08

Approved for public release; distribution unlimited

AFIT/GE/ENG/92D-08

ETANN HARDWARE IMPLEMENTATION FOR RADAR EMITTER IDENTIFICATION

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

James B. Calvin, Jr., A.A.S., B.E.E., M.S.A.
Captain, USAF

December, 1992

Preface

This study applied Intel's artificial neural network hardware, the 80170NX chip known also as the Electronically Trainable Analog Neural Network (ETANN), to classify radar emitter systems. This research study concentrated on comparing simulator results to that of actual hardware implementation for a specific application using collected data.

I want to thank my thesis advisor, Major Steven K. Rogers, for his expert guidance, humor, and encouragement. I also wish to thank Captains Daniel Zahirniak and Gregory Tarr for their helpful support throughout my research project. Special thanks must also go to Captain Dennis Ruck, with whom I spent moments in conversations that significantly enhanced and enlightened my knowledge of C programming. And, finally, all my love and gratitude to my wife, Ruth, from whom I have received encouragement, understanding, and precious time needed to research and write this thesis.

James B. Calvin, Jr.

Table of Contents

List of Figures

List of Tables

Abstract

This study investigated classification of 30 radar emitters with 16 signal features using Intel's 80170NX chip, the Electronically Trainable Analog Neural Network (ETANN). Software tools were developed to characterize the ETANN sigmoidal transfer function for use in a custom simulator, known as Neural Graphics. Neural Graphics operates on a Silicon Graphics workstation. The Intel Neural Network Training System simulators were used in early experiments, but were found to be inefficient in training on data used in this research. Using a modified Neural Graphics simulator, single chip and multi-chip experiments were performed to provide benchmark results prior to performing chip-in-loop training. By maximizing off-chip training accuracy, the need for on-chip training is minimized and therefore the device life is prolonged. Several single chip and multi-chip configurations were tried; the final architecture which produced the maximum on-chip classification accuracy was a hierarchical network. The maximum on-chip classification accuracy for a single chip implementation of 30 classes without chip-in-loop training was 83 percent. Again without chip-in-loop training, the maximum on-chip classification accuracy for a hierarchical configuration with the 30-class problem was 87 percent.

ETANN HARDWARE IMPLEMENTATION FOR RADAR EMITTER IDENTIFICATION

I. INTRODUCTION

1.1 General Issue

Electronic warfare continues to be a primary focus of Air Force research and development. From the simple concept of using radar detectors to warn pilots of possible hostile threats has evolved the idea of identifying those threats, so pilots may gain the advantage by taking active, evasive action. Artificial Neural Networks (ANNs), also referred to as neural networks, now make it possible to explore old ideas of automating emitter identification in cockpits. Because aircraft have limited space and computing capacity, ANNs may be the ideal solution to radar emitter identification. This discussion leads precisely to what this thesis addresses, solving a radar emitter identification problem using ANN hardware.

1.2 Background

An ANN is a model that simulates a biological neural network system; it may model brain processes or brain capabilities. The word neural was derived from neuron. A neuron is a nerve cell that is used to process, store, retrieve, and electrochemically manipulate information received from the cell body or input channels, called dendrites. The human body contains tens of billions of neurons, each linked to potentially thousands of others through a biological network. Each neuron features several inputs, but only one output. To produce an output, the neuron sums the inputs from the dendrites at a common point and then compares the sum against

a threshold value. If the threshold value is exceeded, the neuron activates a pulse voltage in the output. Similarly, an artificial neuron may have many inputs, but only one output. Figure 1 illustrates a biological neuron, and Figure 2 illustrates an artificial neuron, which is also referred to as a single perceptron proposed by Rosenblatt in 1959 (29). The only true similarity between a biological and artificial neuron is the input/output structure and their activation nature. A parallel combination of neurons constitutes a network layer, and one or more layers defines a neural network. Two or more layers is also referred to as a multilayer perceptron network. Moreover, the ANN network may have more than one output, depending on how many decision functions are associated with the ANN output (27:1-15).

The hardware implementation of a complex ANN model makes use of a massively parallel interconnected network of perceptrons (the artificial equivalent of a biological neural network), which interacts in a way similar to a biological nervous system. The ANN accomplishes tasks by modeling a biological system; specifically, the ANN attempts to assign an unknown pattern from the environment into one of a set of selected classes (supervised method) by emulating the biological neural network.

Engineers and scientists are applying the ANN to such tasks as automatic target recognition. As described by Ruck (30), the process of automatic target recognition consists of three stages: segmentation, feature extraction, and classification. Segmentation is the operation of isolating the target of interest from its environment, whereas feature extraction involves processing the data to compute the features that allow discrimination among different target classes. The classification stage assigns each input target to a class based on its features.

It has been demonstrated that neural networks can be used to design radar classification systems on the order of 30 classes with high accuracy. Cameron (5) demonstrated that using a hierarchical approach to classify 30 classes results in improved system accuracy over that of a straight forward application of one network.

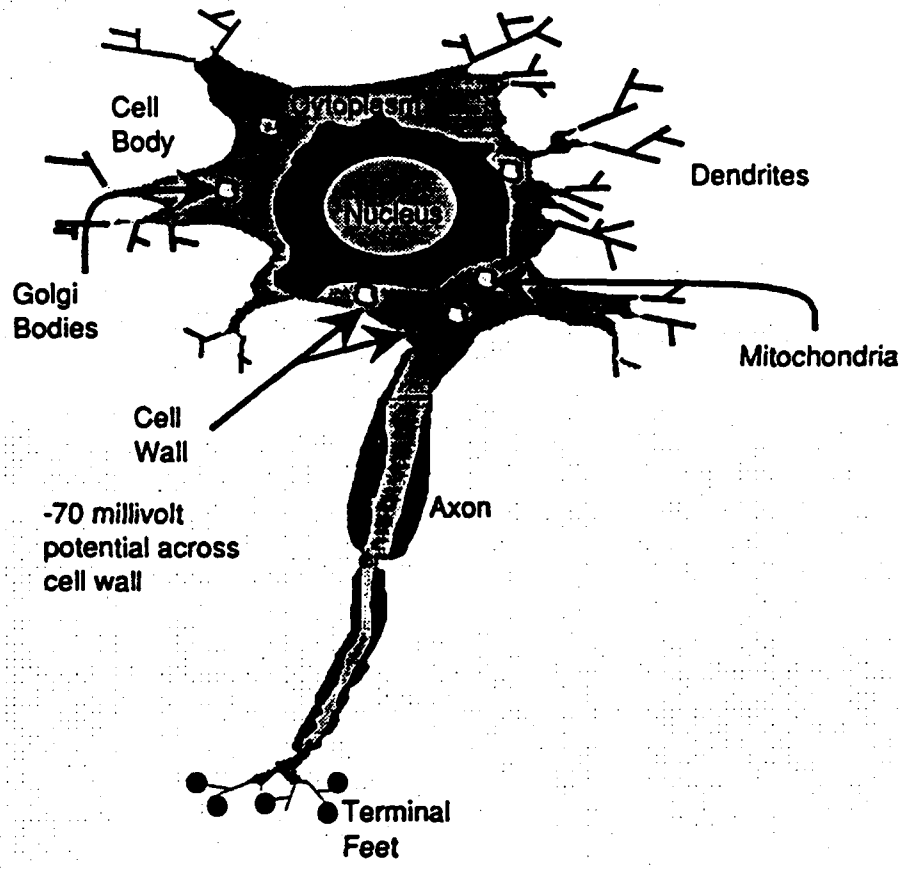

Figure 1. Biological Neuron (28:19)

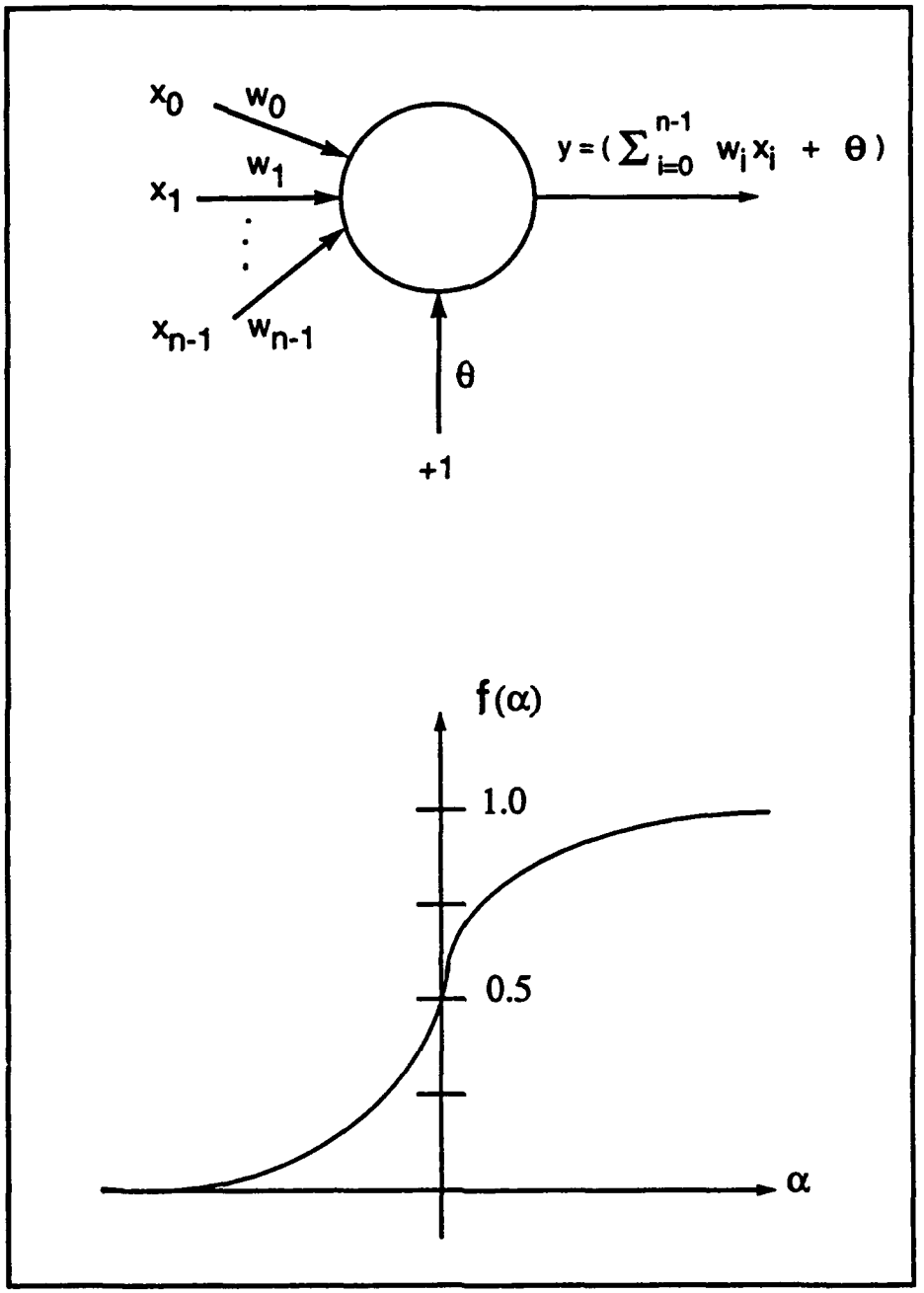

Figure 2. Artificial Neuron or Single Perceptron (28:47)

Using the hierarchical approach, Capt Cameron succeeded in achieving a 96.5 percent accuracy rate of classifying the radar emitter patterns correctly (5:52). A discussion on the hierarchical approach used in this research is covered in Chapter 4.

1.3 Problem Statement

The research problem is to implement a simulated neural network from (5) in hardware and to achieve the accuracy results recorded from simulation. The hardware used in these experiments is built by the Intel Corporation. The primary hardware device is the 80170NX chip, an Electronically Trainable Analog Neural Network (ETANN). In addition to the ETANN chip, Intel built the Personal Computer (PC) interface hardware and software that allows the designer to program the ETANN chip after simulator training (13:1-3).

Originally, the principal investigator (5) was given 32 classes of radar signals to implement in an artificial neural network simulator. Two of the classes had too few of vectors to train effectively; therefore, they were excluded from further experiments, and thus the problem was further defined as 30-classes. Then the Neural Graphics simulator, which was coded in the C computer language at AFIT (37), was used to train on the radar signal emitter patterns. The back propagation training paradigm was used in Neural Graphics.

Little has been said about the data because the focus of this research is on hardware implementation. The data used for training and testing the hardware is real, collected data, intercepted and recorded by the Georgia Technical Research Institute (GTRI). This data was used in previous experiments (5).

The features of this data were extracted by GTRI and transformed into samples of data vectors in American Standard Code for Information Interchange (ASCII) format. Each vector contains 16 elements representing 16 features used for pattern classification. The vector elements are measured features of the intercepted radar signals.

1.4 Research Objectives

The primary research objective is to implement an artificial neural network in Intel's ETANN hardware to classify the GTRI radar emitter data. In addition to the primary research objective, the following sub-objectives will be accomplished:

1. Minimize the processing time and maximize the classification accuracy. This sub-objective will require setting up experiments to prune the network of bad data and to consider chip-in-loop training, which requires using the ETANN chip to train in real time. Minimizing the processing time implies using as few processing layers as possible.
2. Develop a technique to load training weights from the Neural Graphics simulator (37) to Intel's Neural Network Training System (iNNTS). This technique will require transfer of the normalized data as well as the weights.
3. Estimate hardware processing time to make a classification. The processing time in hardware is important information that will indicate how close to real time the processor functions.
4. Compare resolution differences between the hardware device and simulator. Resolution is important because information is usually lost when going from simulator to hardware, which may affect network performance in classifying the radar emitter feature data.
5. Calculate an implementation loss factor. The loss factor is a loss associated with hardware implementation when going from theoretical to actual results. In other words, the loss factor is the difference between theoretical (software simulation) and actual results (hardware implementation).

1.5 Scope

This research focuses on a single application of classifying 30 radar emitters in ANN hardware based on discrimination of feature vectors. To narrow the scope, this

research is exclusively limited to using Intel's ETANN chip as the neural network hardware device. The simulators used in this research are AFIT's Neural Graphics (37) and iBrainMaker (3). The data used in training were limited to a fixed number of feature vectors (a pool of 8,469 vectors).

1.6 Chapter Outlines

In chapter 2, a literature review was accomplished to look at all the information necessary for understanding the radar emitter classification problem and hardware solution approaches. Chapter 3 presents a detail map of the methodology used in the forming of a solution to the research problem. Chapter 4 presents the findings of the research problem and sub-objectives. Finally, Chapter 5 presents conclusions and recommendations.

1.7 Summary

The purpose of this research is to solve a 30-class radar emitter identification problem using Intel's neural network hardware device, the ETANN chip. In addition, a series of tests and comparisons with simulation results were performed in an effort to establish a theoretical versus practical implementation loss factor for this network. This loss factor is a measure by which future simulations can be adjusted for accuracy. Two simulators were used for comparison of training results. The degree of success of this research will help determine future Air Force developmental work in neural network hardware.

II. LITERATURE REVIEW

2.1 Introduction

This chapter reviews literature pertinent to understanding the radar emitter classification problem and solution approaches. Below are the two topics discussed in this chapter:

- Radar Emitter Classification
- Neural Network Hardware

2.2 Radar Emitter Classification

Artificial neural networks are rapidly becoming the favorite algorithm in emitter identification. The speed at which neural networks operate and the robustness of the network architecture make them attractive to electronic warfare engineers. This review of radar emitter classification is intended to document a baseline of parameters achieved in neural network simulations, so that the researcher can compare hardware implementation results of the Electronically Trainable Analog Neural Network (ETANN) to that of simulations in general.

Radar emitter classification results until now have only been documented based on simulation results. Thus far, the best simulation results of a large-class emitter identification classifier were achieved by Capt Cameron at the Air Force Institute of Technology (5). His best test results for a 30-class, 16-feature problem was a 96 percent classification accuracy using a hierarchical network approach (5:52).

In a medium-class problem, Capt Zahirniak at the Air Force Institute of Technology used a kernel classifier (radial basis function) to classify radar emitters. His network was configured for six features on the input and ten classes on the output. Capt Zahirniak achieved an 86 percent classification accuracy (43:5-17).

Willson describes the use of an artificial neural network (ANN) for classifying radar signals collected by a passive receiver (42). He used an ANN employing a multilayer perceptron network using back propagation training to classify radar emitters. Willson's network consisted of six input features and fourteen classes on the output. Using monopulse radar data as features for the input of the network, Willson compared the ANN employing a multilayer perceptron to a template-based technique and a K-nearest neighbor classifier. He achieved an average of 94 percent correct classification using an ANN, an average of 86 percent correct classification using a K-nearest neighbor algorithm, and an average of 74 percent correct classification using a template match.

At the Georgia Tech Research Institute, Howitt used a standard feed-forward network with three input features (RF, PRI, and PW), eight hidden nodes, and ten emitter classes to test the performance of an ANN using the back propagation training algorithm (11:213). Howitt's simulator results for four independent tests using testing data sets (data not shown to the networks before) developed for each test showed an average correct classification of 99.9 percent.

Standard feed-forward networks suffer from several drawbacks. First, network training size is directly related to the size of the network. In other words, as the number of emitter classes increase, the rate at which the network converges decreases (slower convergence). Second, when a network is modified by adding or deleting emitter classes, it has to be retrained (the entire network of all classes). To mitigate these drawbacks, Howitt developed an alternate method to the standard feed-forward network, called the piecewise training network.

Howitt divided the standard feed-forward network described above into two networks, each with three input features as before, three hidden nodes instead of eight, and five output classes instead of ten. The networks were trained individually and "glued" into one final network with two additional hidden nodes. After combining this network, an additional training phase with the entire training data set

Table 1. Radar Emitter Classification

Classes	*Features*	*Type*	*Result*	*Researcher*
10	3	MLP	99%	Howitt
10	6	RBF	86%	Zahirniak
16	6	MLP	94%	Willson
30	16	MLP	96%	Cameron

is required (11:214). For the four simulator tests performed, Howitt found results almost identical to the standard feed-forward network (99 percent average correct classification).

According to Howitt, the advantage of using the piecewise network in this example is that the number of interconnections were decreased from 104 to 74, representing a 30 percent reduction in the number of weights. Fewer weights means less training time required. Moreover, dividing a large network into sub-networks improves convergence time (for separable classes). Last and most importantly, adding or deleting classes can be done more efficiently. For example, the entire network does not require retraining when adding or deleting one class. By retraining only a sub-network in which one modifies an emitter class and then retraining with the "glue" neurons in the entire network, one saves having to retrain the other sub-networks. This method could result in a substantial savings in time when considering large networks.

2.2.1 Summary Table 1 is a summary of the results in current literature found on radar emitter classification using neural networks. In the table, RBF is short for Radial Basis Function, and MLP is short for Multi-Layer Perceptron.

2.3 Neural Network Hardware

2.3.1 Introduction Hardware implementations of Artificial Neural Networks (ANNs) has been a growing field for the last couple of years. Several hardware

models of ANN implementations, also referred to as neural computers, now exist for a number of different training algorithms and network types. This section presents a brief survey of neural network hardware found in current literature.

2.3.2 Survey of Current Neural Computing Hardware

2.3.2.1 DTCNN Chip: Harrer and others have demonstrated an efficient architecture for an analog realization of Discrete-Time Cellular Neural Networks (DTCNN), which take advantage of binary outputs and simple interconnections of several chips. Because this chip was only recently developed, no test performance data of actual realizations have been reported using this chip; however, the chip is being considered for applications such as discrete convolution, connected component detection, concentric contouring, and oscillation (10:466).

2.3.2.2 Neurocomputer Chip: Lont and Guggenbuhl built the Neurocomputer chip that contains 18 neurons and 161 synapses in three layers and provides 16 inputs and 4 outputs (18:457). The complete neural computing system consists of a personal computer for training the network off-chip, an interface card for downloading weights to the chip, and a Neurochip (18:463).

Experiments with the Neurochips have been limited so far; however, the Neurocomputer chip has been tested with a simple pattern-recognition application. It was found that at best, training error can be reduced to a linear error of 11 percent. In portability tests, the authors found that each chip has to be retrained individually because no two chips are exactly the same (18:463). The propagation time delay from the input to the output of each chip is equal to 4 μsec. The overall computation rate (weights times inputs) is 40.25 million connections per second. Lont and Guggenbuhl state that this computation rate is not the effective computation rate because the weights must be refreshed from time to time (18:464).

2.3.2.3 ANNA Chip: Sackinger and others designed, fabricated, and tested a reconfigurable ANN chip, called ANNA (for Analog Neural Network Arithmetic and logic unit) (31:498). The ANNA chip is optimized for locally connected, weight-sharing networks and time-delay neural networks.

The ANNA chip implements 4096 synapses with 6 bits of resolution for each state (input/output of the neurons). All input and output of the chip are digital, whereas internally the chip uses analog computing. The advantages of using the ANNA chip over most other implementations are high synaptic density, high speed, low power requirements, and ease of interfacing to a digital system (31:498).

The processing speed on the chip, from start to finish, takes about 200 nsec (four clock cycles). The ANNA chip is configured for weight vector sizes of 64, 128, and 256, which also correspond to the number of synapses per neuron (31:498). For the maximum configuration size of 256 synapses per neuron, the chip can evaluate a maximum of 10 billion connections per second. For practical applications, however, the speed will be lower because some applications will not make full use of the ANNA chips's parallelism (31:499).

The locally connected, weight-sharing neural network (the neural network designed on the ANNA chip) has been successfully used for optical character recognition of both digits and letters, for character recognition from touch screens, for image segmentation, and for speech recognition (31:498).

2.3.2.4 TInMANN Chip: Melton and others have designed the architecture, operation, and implementation for The Integer Markovian Artificial Neural Network (TInMANN) chip, which features a massively parallel all-digital, stochastic architecture (21:375). The TInMANN uses a modified Kohonen algorithm to simplify the computations on chip. The integer Markovian learning algorithm replaces multiplications and complex function evaluations with simple additions, comparisons, and random number generation (21:377).

The current implementation model uses one neuron per chip, which can take vectors up to six dimensions at 10-bits precision. This model is capable of processing up to 195,000 three-dimensional training vectors per second (21:382). The TInMANN was designed to be flexible and capable of being expanded to larger systems. The primary drawbacks with the TInMANN chip are the low level of integration (only one neuron per chip), the higher power and reliability problems as compared to other Very-Large-Scale-Integration (VLSI) chips, and the slow speed of the network (21:383).

2.3.2.5 ETANN Chip: Intel has developed an Electronically Trainable Analog Neural Network (ETANN), the 80170NX chip. The 80170NX chip is an electronic device that performs ANN functions in hardware, and is currently the most capable ANN device developed so far.

The ETANN chip is a 64-neuron, 10,240-synapse electronically trainable parallel data processor, known as an analog neural network. It is configured for mapping 128 inputs, of which 64 are feedback inputs, into 64 neuron outputs. The ETANN chip features a low power requirement (5-volts operation), a resolution of 7 to 8 bits, a dense 208-pin PGA (Pin Grid Array) package, and a non-volatile weight storage capability (minimum data retention lifetime of 10 years) (13).

A single ETANN chip can perform more than 2 billion multiply-accumulate operations (connections) per second. Moreover, processing throughput takes about 6 μsec (or 3 μsec per layer), near real-time for pattern recognition applications. By interconnecting eight chips, systems can achieve more than 16 billion connections per second, a performance level that exceeds most supercomputers. Hence, this capability makes the ETANN chip a prime candidate for real-time pattern recognition and signal processing applications (13).

Mumford, Andes, and Kern have designed a neural network processing system that incorporates an ANN as a subsystem in a layered hierarchical architecture

(22:423). The ANN hardware used in the system design is Intel's ETANN chip. The Mod 2 Neurocomputer, which is being built at the Naval Weapons Center, is designed to be a neural network processing system, using separate individual neural networks as subsystems in a flexible, modular architecture (22:423).

The Mod 2 Neurocomputer is a reconfigurable, general-purpose neural processor that can be easily tested in an interactive manner and integrated with an imaging sensor for real-time integrated processing demonstrations (22:424). The designers have not completed hardware construction at this time. When completed, the Mod 2 Neurocomputer will be a demonstration of ANN hardware integration into a system designed to support parallel processing of image data at real-time rates.

2.4 Conclusion

This chapter reviewed literature on the topics of radar emitter classification and neural network hardware. The best radar emitter classification accuracy for a large-class problem was achieved by Capt Cameron at AFIT. He achieved a 96 percent classification accuracy on a simulator using the same data that is to be used in this research.

A survey of neural network hardware covered five newly developed neural computer devices: the DTCNN chip, the Neurocomputer chip, the ANNA chip, the TInMANN chip, and the ETANN chip. This survey concluded with an example of a system integration using Intel's ETANN chip. Some of the chips are still in the construction phase and have not been tested, whereas others have undergone limited testing. Besides being the oldest device surveyed, the ETANN chip remains as one of the most capable chips in use today.

III. METHODOLOGY

3.1 Introduction

This chapter is divided into three sections: ETANN description and theory of operation, phase one research, and phase two research. Each phase of research covers tests and analysis that require development of software tools and procedures for implementing the ETANN device for radar emitter classification.

Phase one research tests require performance evaluations of computer simulations, hardware characterizations, and network analysis of a 16-feature, 3-class radar emitter identification problem. Phase two research requires solving several 16-feature, multi-class radar emitter identification problems in hardware—some of the same problems Capt Cameron solved in simulation (5:18). In phase one research, two computer simulators are used for off-chip training; they are briefly discussed below. Phase two research uses only one simulator for the final hardware implementation.

3.1.1 Software Simulators The two simulators used in this research are Neural Graphics (37)(38) and iBrainMaker (3). Each simulator is used for a comparison of speed and accuracy.

The Neural Graphics simulator was developed by Greg Tarr and was used in his doctoral dissertation for research and modelling of neural networks. Neural Graphics implements a neural network model by using a data file containing both training and test vectors, also known as training and test patterns. Neural Graphics features several paradigms for training various kinds of networks. Its basic back propagation paradigm was modified to emulate the ETANN transfer function and is used in this research (see Appendix F).

The Neural Graphics simulator provides several measures of network accuracy while performing the classification task. Most importantly, it measures a percentage

right and *good* for both training and test data, which are updated according to the number of iterations set in the screen update. It is advised that the updates be set to reflect training epochs, where one epoch is equal to the number of training vectors in the data file that contains both training and testing vectors. The term *right* classification refers to one in which the correct output node value is greater than 0.9 and all other nodes have values less than -0.9. The less strict *good* measure of classification is defined as the result from a maximum pick and is used throughout this thesis, termed the *percentage correct* Note: These values were later changed to 0.89 and -0.94 for emulation of the ETANN device, enhancement of training, and reduction of total error (more about this in Chapter 4).

The BrainMaker simulator (v2.02), which was developed by California Scientific, is the only commercial simulator used in this research. The iBrainMaker is the Intel version of BrainMaker adapted for interfacing with the ETANN chip programming software, known as Intel's Neural Network Training System (iNNTS).

3.2 ETANN Chip Description

Intel developed an Electronically Trainable Analog Neural Network (ETANN), the 80170NX chip. The ETANN chip is a neural computing device that performs ANN functions in hardware and is the primary device used in this research. A brief description of the ETANN follows below.

The ETANN chip is a 64-neuron, 10,240-synapse electronically trainable parallel data processor, known as an analog neural network or analog neural computer. It is configured for mapping 128 inputs, of which 64 are feedback inputs, into 64 neuron outputs. The ETANN chip features a low power requirement (5-volts operation), a weight resolution of 7.5 bits (20)(about three decimal places of accuracy), a processing resolution of 6.5 bits (20) (between two and three decimal places of accuracy), a dense 208-pin PGA (Pin Grid Array) package, and a non-volatile weight storage capability (minimum data retention lifetime of 10 years) (13).

A single ETANN chip can perform more than 2 billion multiply-accumulate operations (connections) per second. Moreover, processing throughput takes about 6 μsec (or 3 μsec per layer) for a two layer network, which is near real-time for many pattern recognition applications. By interconnecting eight chips, systems can achieve more than 16 billion connections per second, a performance level that exceeds most supercomputers. Hence, this capability makes the ETANN chip a prime candidate for real-time pattern recognition and signal processing applications (13).

3.2.1 ETANN Theory of Operation The ETANN chip was designed so that most "learning" is performed off chip. Off-chip learning maximizes flexibility as well as speed and enhances the chip's life by reducing on-chip learning time (3:19-8). Chip-In-Loop (CIL) training is performed on the chip after the network has already been trained by the simulator to peak performance. CIL is used to compensate for chip variations and loss of resolution when going from simulator to chip. This process of CIL training is the last step before completing the training process.

The ETANN chip's normal operation can be described as Parallel Distributed Processing (PDP), whereby each of the 64 neurons implements the following function (13:7):

$$\begin{aligned} Output(U_i) &= Sigmoid\{\sum_j Weight(w_{ij})Input(x_j) + \sum_k BIAS_{ik}\} \\ &= \frac{2}{1 + e^{\sum(synaptic\ contributions)}} - 1 \end{aligned} \tag{1}$$

where i equals the neuron number, j equals the number of inputs to neuron i, and k equals the number of biases to neuron i. Although the ETANN chip has 16 bias units per neuron, only biases 0 through 6 are used in normal chip operation. Bias units 10 through 15 are used by the iNNTS software for zeroing the sigmoidal response of neurons.

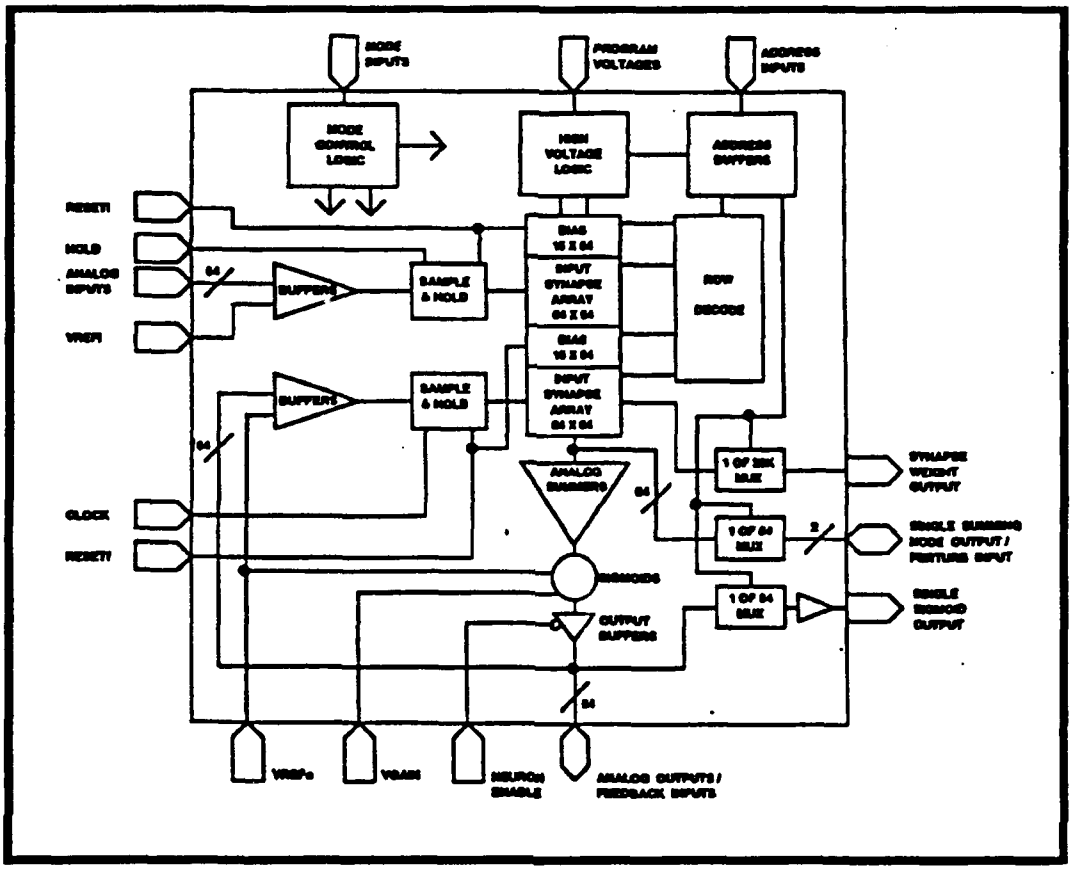

Figure 3. ETANN Block Diagram (13:2)

The equation above represents the inner product operation that is "squashed" by a sigmoidal transfer function. Figure 3 illustrates a block diagram of the ETANN chip. There are 32 fixed-input bias weights and 128 variable input weights that comprise 160 synapses per neuron. Each neuron produces a single scalar output (13:8).

The ETANN chip's processing resolution of the analog inputs and outputs is greater than or equal to 6 bits (about two to three decimal places of accuracy). Typically the weight resolution is at about 7.5 bits (or about three decimal places of accuracy). At worst over a lifetime of 10 years, the processing resolution would be greater than or equal to 4 bits. If the bake/re-training method is used as described

in (13), the worst processing resolution over a lifetime of 10 years would be greater than 6 bits. The bake/re-training method process involves baking a chip in an oven heated to 250°C for 24 hours and then re-training it. According to Intel, the bake/re-training method significantly improves the ETANN weight retention over extended periods of time.

The weights on the ETANN chip are stored as analog transconductance values. Each synapse produces an analog output current from an analog input voltage and a stored weight voltage. The currents generated by each of 160 synapses along a "dendrite" are summed to form an input to a neuron. This sum of currents is then converted to a voltage and used to activate a sigmoid function, which has a voltage-controlled gain. When in the 128-input mode, the input and feedback arrays are active and the output corresponds to the sum of their two dot products.

Figure 4 illustrates the synapse operation on the ETANN chip. Each synapse contains a multiplier and a stored weight, which are used to calculate one product with an input. All the single products are summed from other synapses to produce the dot product (13:9).

3.2.1.1 ETANN Chip Processing Configurations Figure 5 illustrates single ETANN chip processing and shows two basic configurations: (a) a configuration where 64-dimensional input vectors are directly mapped into 64-dimensional output vectors, and (b) a configuration where 128-inputs are mapped into 64 outputs by multiplexing the analog input pins I_0–I_{63} to the feedback array. Pin A_{12} determines the mode that the ETANN chip is in, either multiplex mode or direct mode.

Figure 6 illustrates two more chip configurations: (a) a two-layer processor, and (b) a Hopfield network. These configurations are also possible with a single ETANN chip. Several other configurations are published in (13).

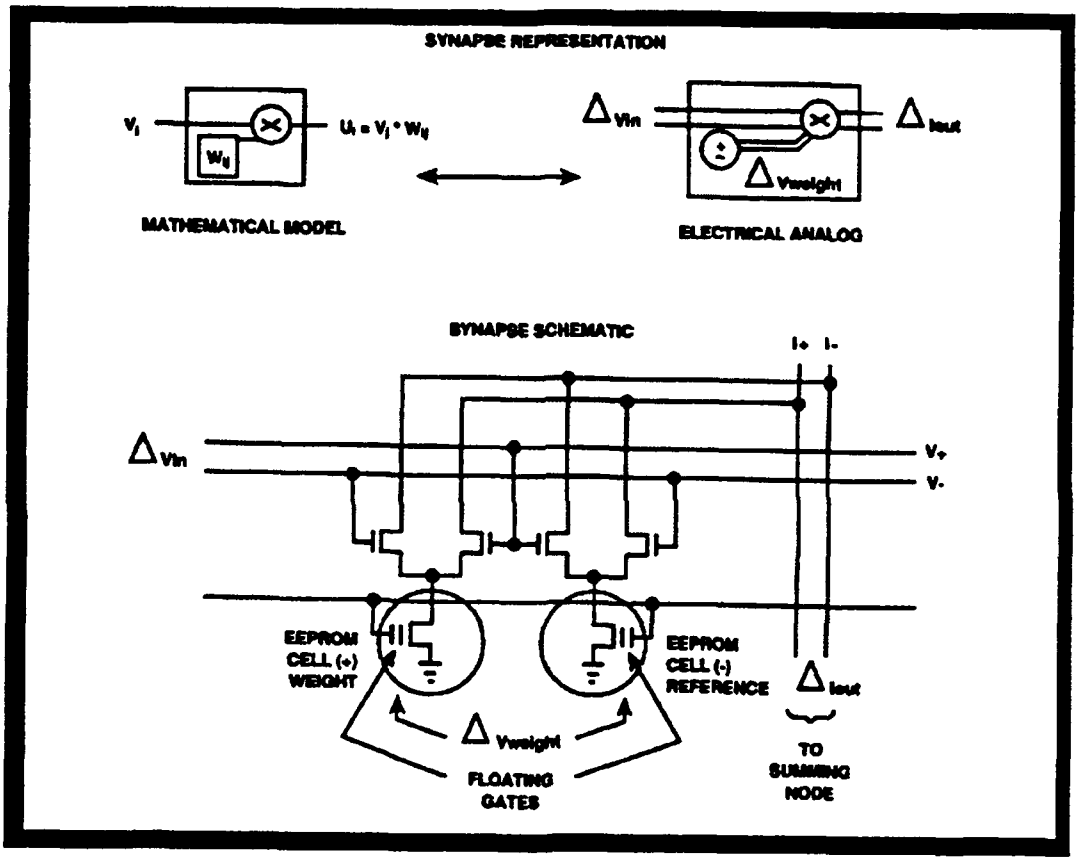

Figure 4. ETANN Synapse Implementation (13:9)

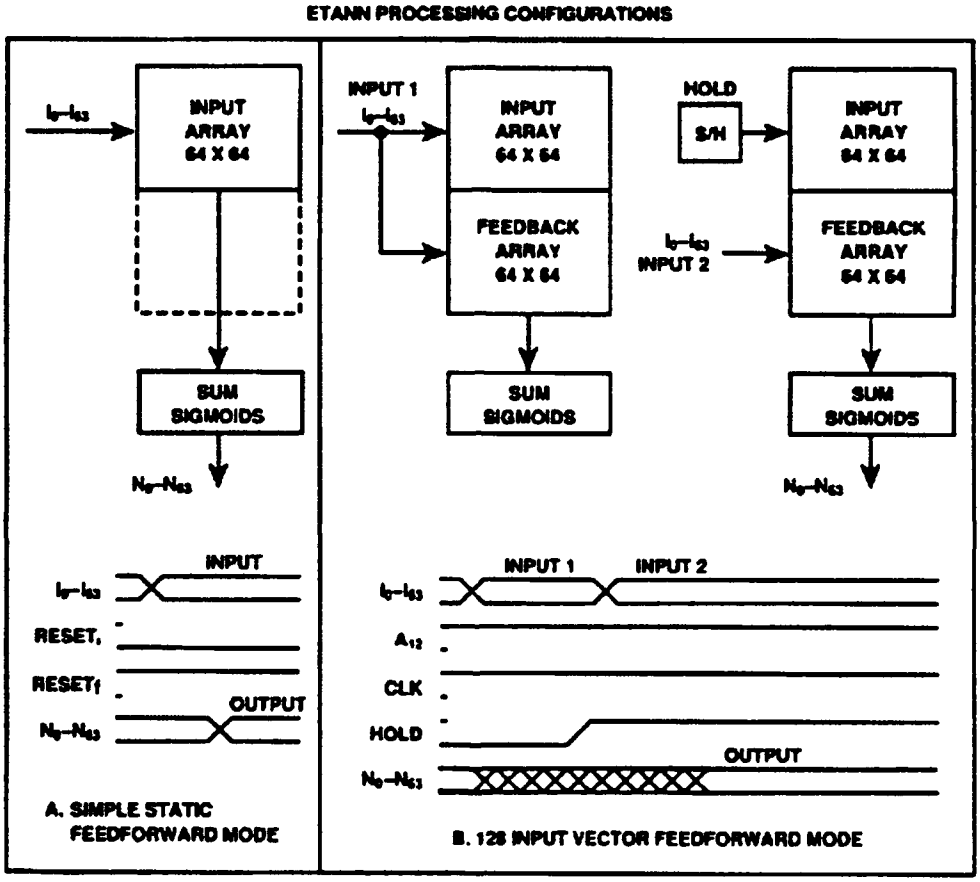

Figure 5. ETANN Processing Configurations (13:11)

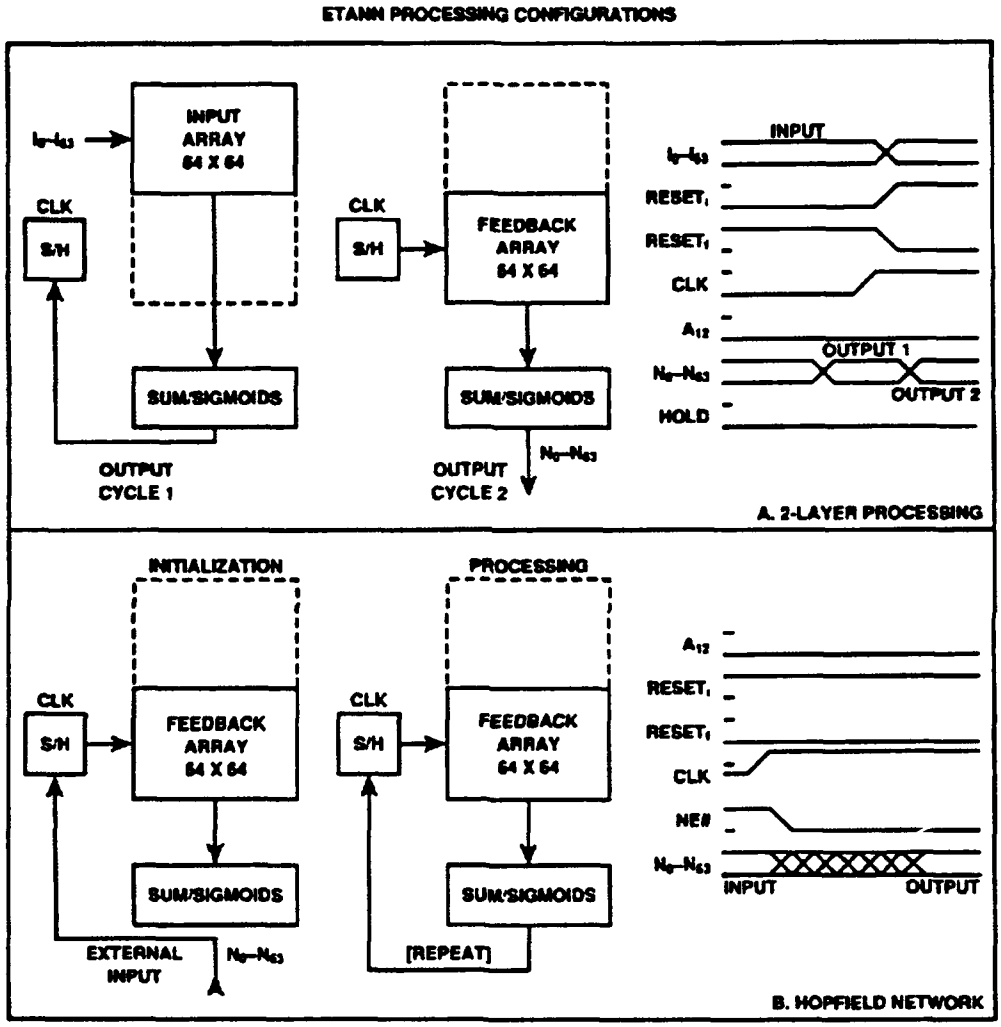

Figure 6. ETANN Processing Configurations (13:12)

3.2.1.2 ETANN Multi-Chip Processing Configurations Intel provides two basic multi-chip processing configurations: 1) Direct Pin Interconnect, or 2) Bus Interconnect. The more straightforward method is direct pin-to-pin because no multiplexing is required. Figure 7 illustrates both multi-chip configurations. Although direct pin-to-pin is simpler, bus interconnection provides more flexibility. Bus interconnection, however, has two drawbacks: 1) data can be moved only in 64-neuron blocks, and 2) multiplexing is slower than direct pin-to-pin due to clocking.

3.2.2 ETANN Transfer Function Characterization Intel published the following sigmoidal transfer function characteristics that simulate the ETANN chip at what Intel termed as typical operating conditions (for binary outputs): $V_{GAIN} = 5V$, $V_{REFo} = V_{REFi} = 1.4V$, for 64 input operation (13:27). V_{GAIN} sets the gain of the device sigmoid; it can essentially change the sigmoid from a linear function over the input range to a step function at the high end (4:9). V_{REFo} and V_{REFi} define the null voltage levels independently for inputs and outputs.

$$Neuron\ Output = \frac{2}{1 + e^{-8\sum x_i w_i}} - 1 \tag{2}$$

Intel's more accurate model (for binary outputs) that includes roll-off due to large input and weight magnitudes as well as output range limitations is shown below (13:27).

$$Neuron\ Output = \frac{1.8}{\left(1 + e^{-8\left(\sum x_i(1.2-0.2x_i^2)w_i(1.5-0.5wi^2)-BIAS\right)}\right)} - 0.9 \tag{3}$$

where x_i are the input values and w_i are the weight values. Inputs are constrained between $-1 \leq x_i \leq 1$. Weights are constrained between $-2.5 \leq w_i \leq 2.5$. For

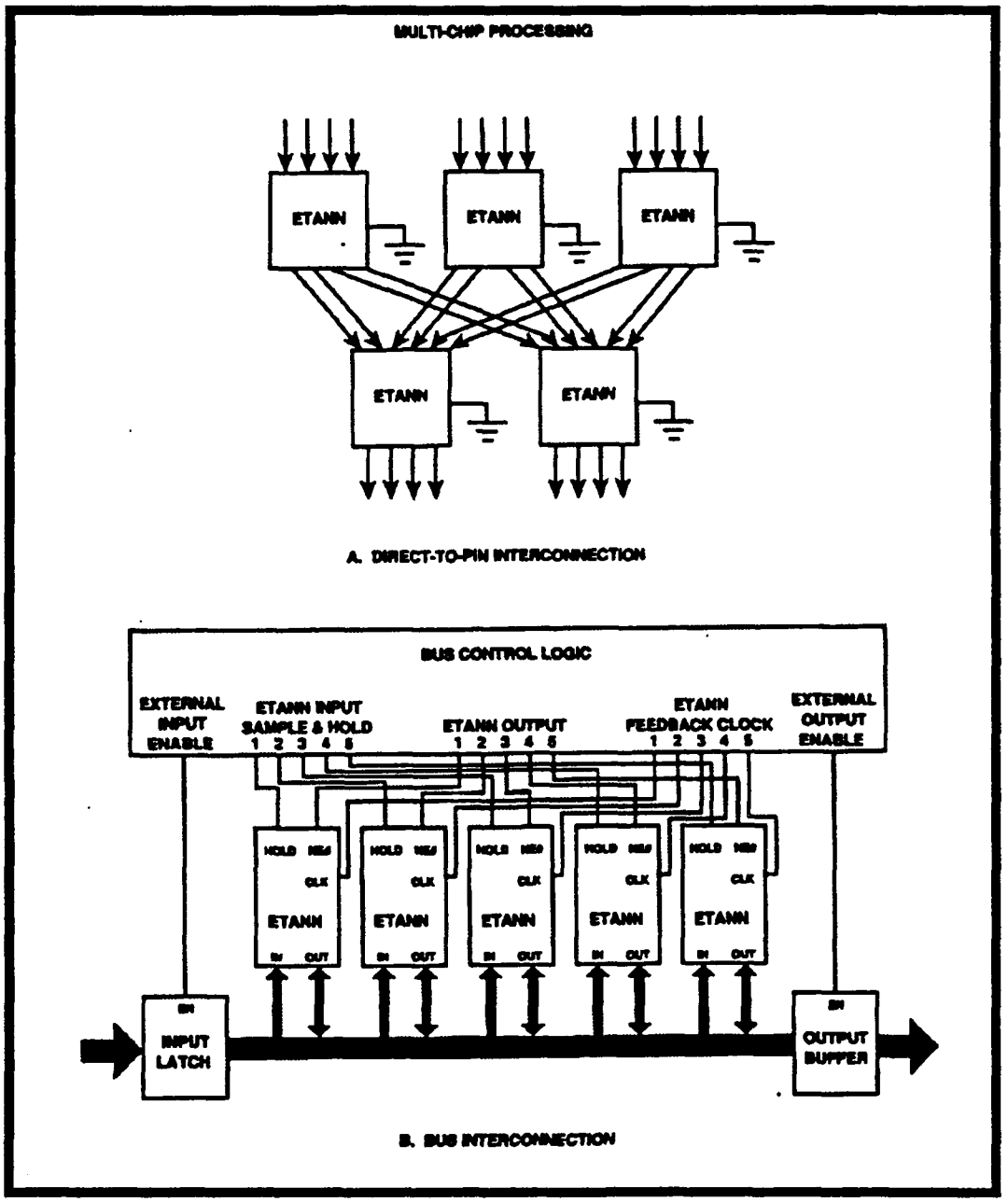

Figure 7. ETANN Multi-Chip Processing Configuration (13:15)

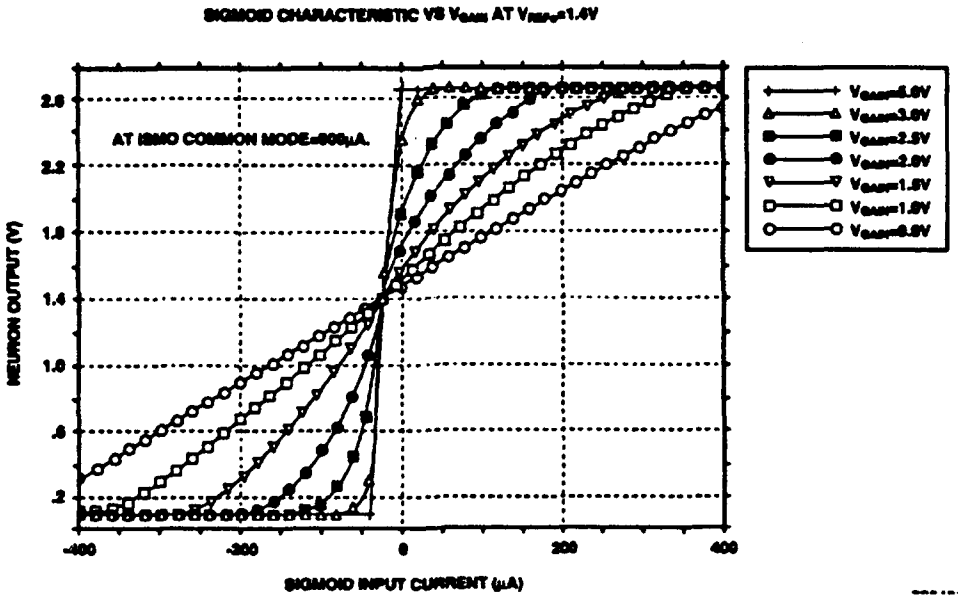

Figure 8. Neuron Output Transfer Characteristics Versus Summing Current and V_{REF} (13:25)

non-binary output operation, Intel has not derived an accurate sigmoidal transfer function, which is the impetus for characterization experiments.

Figure 8 illustrates the neuron output transfer characteristics versus the summing current and V_{GAIN} at $V_{REFo} = 1.4V$. Each of the curves represents the neuron output given a particular V_{GAIN} value.

The Neural Graphics simulator was configured with the transfer function in Equation 2, but the network failed to train properly. When the weights were exported from Neural Graphics and downloaded on the ETANN chip, the result was poor classification. The cause of this discrepancy was not clear until the discovery that the V_{GAIN} for the published transfer function was set at $5.0V$ (used for binary output operation only), whereas the V_{GAIN} for normal sigmoid operation on the ETANN is set to $3.30V$. Consequently, the operating sigmoidal transfer function was not accurately known for a $V_{GAIN} = 3.30V$.

According to (3), iBrainMaker uses a lookup table as opposed to a pure mathematical transfer function to emulate the ETANN sigmoid. A transfer function was extrapolated by the researcher from the menu information given in the program. Given in the menu are the high, low, and gain values for the sigmoidal function. Using this information, the look-up table (resolution unknown) probably emulates the following sigmoidal transfer function:

$$iBrainMaker\ Neuron\ Output = \frac{1.765}{1 + e^{-1.0 \sum (synaptic\ contributions)}} - .915 \qquad (4)$$

Note: The iBrainMaker sigmoidal transfer function was extrapolated from the high, low, and gain values *after* performing independent characterization experiments in this research (see Chapter 4 results).

Researchers who desire to interface their own custom simulator with the ETANN hardware are encouraged to characterize the ETANN sigmoidal transfer function through experimentation. Chapter 4 presents a method used in this research to characterize the ETANN sigmoidal transfer function along with findings from this research. It's worth mentioning here that chip-in-loop training if used excessively, decreases the life of the chip (3); therefore, application designers should optimize training off-chip so as to minimize on-chip training.

3.3 Phase One Research

The purpose of phase one research was to get software and hardware tools working to solve a small network problem, and then use the knowledge learned from phase one to solve more complicated problems in phase two. Phase one research was essentially a developmental step to solving the thesis problem as outlined in Chapter 1.

The test network constructed for hardware implementation consisted of 16 features and 3 classes. The number of feature vectors used in phase one research was 600. This number ensured that Foley's rule was not violated. Foley's rule states that the number of training vectors per class should be greater than three times the number of features (27:61). When this criteria is met, the observed error rate on the training data is a good predictor of the network error rate on an independent set of test data. As long as the total number of vectors per class is greater than 48 for a given 16-feature training set, Foley's rule will not be violated. From the total training set of 600 vectors for this test problem, each class had 200 vectors, which easily satisfied Foley's rule.

3.3.1 Equipment Required The required equipment for this research included a Silicon Graphics workstation for running the Neural Graphics software simulations and Intel's Neural Network Training System (iNNTS) for interfacing with the ETANN chip. The iNNTS includes an Intel 486 personal computer, a Generic Programmer Interface (GUPI), a Personal Computer Personal Programmer (PCPP) board which links the PC to the GUPI, an 80170NX adapter that connects the ETANN chip to the GUPI, and a software simulation program to interface with iNNTS (iBrainMaker). The data for training and testing the ANN was provided by the Georgia Tech Research Institute (GTRI), Atlanta, Georgia.

3.3.2 Software Development GTRI supplied the data in an ASCII code format for this research; however, the file format was incompatible with the simulator formats used in this research. Consequently, the GTRI data had to be reformatted for each simulator, shuffled in a file, and statistically normalized.

This task was broken down into three steps:

1. Transform the GTRI data into Neural Graphics and iBrainMaker formats. Two C programs were written to accomplish this first step and are included in Appendix D: "gtri_to_ng.c" and "gtri_to_bm.c".

2. Shuffle the lines of vectors for both Neural Graphics and iBrainMaker formats. Two C programs were written to accomplish this second step and are also included in Appendix D: "shuffle.ng.c" and "shuffle.bm.c".

3. Statistically normalize the data. The type of normalization used was a normal Gaussian distribution that was "squashed" (between -1 and 1). The ETANN chip must be trained and tested by data dynamically normalized between -1 and 1. Two C programs were written to convert the data to two simulator formats discussed previously and are also included in Appendix E: "norm.ng.c" and "norm.bm.c".

3.3.3 Procedure The network design procedure is basically a seven-step process of working through a logical description of the problem to be solved. In phase one, the problem was to characterize a 16-feature, 3-class problem. The first step in the design process is to determine whether the network is to predict, generalize, or recognize. From the research problem presented in Chapter 1, the network was supposed to recognize radar emitters. The second step requires a designer to choose information that will make good features for recognition . GTRI performed this step and the following step, which was to get lots of data.

Generally, the more data, the better the predictor, generalizer, or recognizer. In some cases, too much data will overwhelm the network. A good method of determining over training is to plot the total error for the network versus the number of training iterations or epochs. When the the curve begins to reverse direction (higher error), it is time to stop training.

The next step is to build the network. When building a feed-forward network, there are two things needed to make the network. One needs samples of input data (training vectors) and to know the desired results.

Two things are needed to make a network: a network definition and a collection of training vectors (34). A network definition consists of assigning a number of input nodes to a number of hidden nodes and a number of output nodes. The assigning of input nodes and output nodes is problem dependent. The assigning of hidden nodes is problem and data dependent. If the problem is linearly separable, then hidden nodes are not necessary. The goal is to train effectively with as few hidden nodes as possible, so as to minimize the number of computations in the network.

After building the network comes training the network. Training a network is more an art than a science. Adjusting learning parameters such as momentum, learning rate, and tolerance all become factors in how fast a network will converge, if it ever does.

Testing the network follows training the network. A percentage of the data is always saved as testing data. Generally, 10–20 percent of the total number of vectors is sufficient to test the network. It is important to show the network data that it has not seen before, because only then will one know if the network is a good predictor, generalizer, or recognizer. The last step in this process is running the network, presenting it with new input data, and gathering usable results. At this point, one has completed the seven-step design process.

3.4 Phase Two Research

In phase two, a solution to the main problem of characterizing 30 radar emitters in hardware was the focus of research and analysis. Using what was learned in phase one, the task was simplified to that of using the same basic tools and procedures.

3.4.1 Equipment Required The required equipment for phase two research was the same as that described in phase one with the addition of using the ETANN Multi-Chip Board (EMB) for multi-chip prototyping. The data for training and testing the networks in phase two were an extension of what was used in phase one, which included all 30 classes.

3.4.2 Software Development Software developed in phase one research was extended in phase two for construction of larger networks. The programs used in phase one were modified to work in phase two. The main difference in the programs is the handling of more classes and pattern vectors.

3.4.3 Procedure The procedures for designing an artificial neural network are basically the same as those discussed in phase one, but the problem was much more complicated by the fact that the number of classes was increased. Capt Cameron published a baseline of test findings that are used here as a comparison with hardware implementation results (5).

3.4.4 Baseline Tests Capt Cameron performed a series of experiments with the Neural Graphics simulator to provide a baseline against which to compare experimental results. Accordingly, the number of classes per data file and the number of vectors per class were both varied as widely as possible. In addition, the class numbers were reassigned to ensure a random combination of classes. The transfer function used in Cameron's simulation was a sigmoidal function equal to $1/(1+e^{-\alpha})$ over the interval [0,+1]. The ETANN chip is characterized by another sigmoidal transfer function, which will be discussed in Chapter 4. The baseline of experimental results is included here with note of the following information:

- Classes: randomly selected, numbers as indicated in Table 2
- Training vectors: as indicated in table
- Test vectors: 25 per class

Table 2. Baseline Testing (5:18)

Training		Percentage Correct (Test/Train)				
Vectors	Iterations	Number of classes				
(per class)	(x 1000)	8	14	20	26	30
25	10	96/98	96/95	84/88	84/83	70/70
25	20	97/100	96/98	88/91	88/89	87/89
25	30	98/100	96/99	91/94	90/91	88/90
50	10	96/99	94/96	90/92	88/90	82/80
50	20	96/99	95/98	89/92	92/94	90/92
50	30	95/100	95/99	91/96	92/95	89/92
75	10	96/97	97/97	86/88	82/83	insuf. data
75	20	97/99	97/98	89/92	92/93	insuf. data
75	30	97/99	98/98	91/94	92/93	insuf. data
100	10	98/100	96/97	87/90	78/79	insuf. data
100	20	98/100	97/97	92/93	89/90	insuf. data
100	30	98/100	97/98	91/95	92/93	insuf. data
125	10	98/99	96/96	90/91	76/77	insuf. data
125	20	98/99	96/98	90/91	88/90	insuf. data
125	30	99/100	97/98	93/95	91/92	insuf. data

- Hidden nodes: 16
- Iterations: as indicated
- Special note: some classes have too few vectors; thus this is why there are *insufficient data* entries

In Table 2, it is evident that classification accuracy increases as the number of training vectors increase, and the accuracy decreases as the number of classes increase, as expected. Accuracy generally increases as the number of iterations increase, but the effect is more pronounced for runs with twenty or more classes. This result is expected since the larger networks had to be trained with more vectors. The maximum baseline accuracy recorded for a single network, 30-class problem is 90 percent. Of course, this result is not Capt Cameron's best result. Using a hierarchical method, he achieved a 96 percent accuracy with an 18(top)/12(bottom) split of classes in 8,469 test vectors representing half of each class (5).

3.5 Conclusion

This chapter has given the reader a description of the ETANN chip, a synopsis of phase one and two research objectives, and a development plan for software tools and procedures needed for implementing a neural network device to solve a radar emitter identification problem. In addition, a baseline of experimental results from (5) was included here as a bench-mark for device implementation.

IV. RESULTS

4.1 Introduction

This chapter presents the experimental results for the problems that were discussed in Chapter 1. Phase one research findings include the sigmoidal transfer function characterization results, simulator test comparisons, and hardware implementation tests. Phase two research findings include single chip and multi-chip implementations using Intel's ETANN Multi-chip Board (EMB).

4.2 Phase One Research Findings

4.2.1 ETANN Sigmoidal Transfer Function Characterization Experiment The general procedures for the ETANN sigmoidal transfer function characterization experiment are outlined as follows:

1. Interface with Intel's Neural Network Training System to perform read and write functions on the chip's inputs and synapses.
2. Prepare the ETANN device for programming by using the "prepare chip" command in iNNTS—use the following settings: $V_{GAIN} = 3.3V$ and $V_{REFi} = V_{REFo} = 1.5V$, 64-input operation, number of initialization biases = 9, and weight precision set to 0.042.
3. Perform a series of tests on a number of available chips (eight were used for this experiment) over a range up to the maximum input values (input values = -1.0, -0.75, -0.5, -0.25, 0.0, 0.25, 0.5, 0.75, 1.0) and weight values (weights values= -2.5, -2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.5) in order to calculate the average ETANN chip response.
4. Write input values to a single ETANN neuron with varying numbers of inputs (e.g., 1, 2, 16, and 32).

5. Calculate the average hardness parameter (a sigmoid gain coefficient) for each experiment of varying inputs (hardness parameter equals eight in Intel's characterization equations).
6. Plot activation values versus output values to characterize the sigmoidal transfer function of the ETANN chip.
7. Write a program to calculate the mean-square-error (MSE) versus the hardness parameter changes, and plot the results for various number of inputs to a single neuron.
8. Develop a mathematical model for the transfer function, and code the function in Neural Graphics (or your favorite simulator).
9. Train a network using a simulator with the newly characterized sigmoidal transfer function, and then download the weights to an ETANN device for on-chip testing.

4.2.2 ETANN Sigmoidal Transfer Function Characterization Findings Programming in C was necessary for weight format conversion between Neural Graphics and Intel's Neural Network Training System (iNNTS). In addition, several C programs were written to analyze and process data collected from iNNTS; a sample of these programs may be found in Appendix G.

Figure 9 illustrates a data fit to the sigmoidal transfer function, which was characterized by averaging data collected from eight ETANN chips. In each of the plots, the x-axis is called the activation, and the y-axis is called the output. The activation represents a sigmoidal function acting on the summation of all inputs multiplied by associated weights plus biases. The biases in this experiment were set to zero during the prepare chip step. The inputs were written to a single neuron (all tests were performed on a single neuron) in incremental steps (input values = -1.0, -0.75, -0.5, -0.25, 0.0, 0.25, 0.5, 0.75, 1.0). The weights were changed over a range of 11 values from maximum negative to maximum positive values (weights

values= -2.5, -2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.5). For example, the first weight value chosen, -2.5, was fixed and then the inputs (depending on how many were selected for that particular test) were written to a single neuron, incrementing through the entire range of nine input values. Then the weight was changed (set to -2.0, the next value). The inputs were written to a single neuron again, incrementing through the entire range of nine values. This process was repeated for all 11 weight values.

Additionally, each plot in Figure 9 shows the result of a particular test (test1, test2, test3, and test4 as labeled on the plot headings). Test1 represents a single input to neuron zero of each of the eight ETANN chips. Test2 represents two inputs to neuron zero of each of the eight ETANN chips. Test3 and test4 represent 16 and 32 inputs respectively to neuron zero in each of the eight ETANN chips.

For each test, a different hardness parameter was found by searching for the minimum MSE over a range of various hardness parameter values between zero and nine. Table 3 shows the hardness characterization experiment results for a $V_{GAIN} = 3.3V$ and a $V_{REFo} = V_{REFi} = 1.5V$ (64-input operation). For each activation value, a numerical output was automatically measured and recorded in a data file along with the summation of inputs multiplied by the weight value (sum of products), which was fixed for the entire range of inputs. This process was repeated until all 11 weight values were used. The plots in Figure 9 illustrate activation versus output (99 points in tests one and two and fewer in three and four due to limiting of the x-axis to [-8:8]). The averaged data represents the average activation response from eight ETANN chips for the nine input values and 11 weight values. Tests three and four results show erratic behavior near zero activation values (see plots C and D). The behavior is a reaction to changing weight values (over the 11 values mentioned above) with zero (or near zero) input values. This erratic behavior might be explained by noting that as more inputs are added to a neuron, a larger current is drawn and thus the transfer function characteristics change—modelling such change is difficult if not

Table 3. Hardness Characterization Results

Experiment	*Number of Neuron Inputs*	*MSE*	*Hardness Parameter*
1	1	0.00412	1.7
2	2	0.00355	1.5
3	16	0.00449	1.2
4	32	0.01235	0.9

impossible for all network variations; therefore, an average value for these points was taken as the overall slope coordinate.

In Figure 10 the hardness parameter versus MSE is plotted over the range [0.0:9.0] for all tests. In training experiments using the four different hardness parameters found in the MSE tests, neural networks for this problem trained to a lower total error while using the largest hardness parameter (from experiment one) to characterize the transfer function. Tests two, three, and four show that the transfer function's output slope decreases slightly with more inputs, but the summation of synaptic contributions usually exceeds the dynamic range where the hardness parameter has the most effect on the output. For example, if a network has 16 inputs, the dynamic range of the activation function parameter α (sigmoid[α]) is between $-40 \leq \alpha \leq 40$, where α is the summation of weights times inputs (assuming biases are set to zero in this example). The sigmoidal function's slope region is most pronounced between $-2 \leq \alpha \leq 2$, which is a very small region compared to the entire dynamic range. As the number of inputs increase, the transfer region becomes less significant in the sigmoid output. Therefore, using the largest hardness parameter measured for all network sizes should give the best result for simulation without having to adapt the hardness parameter for different sized networks.

4.2.2.1 ETANN Sigmoidal Transfer Function Based on experimental results for a $V_{GAIN} = 3.3V$ and a $V_{REFo} = V_{REFi} = 1.5V$ (64-input operation), the

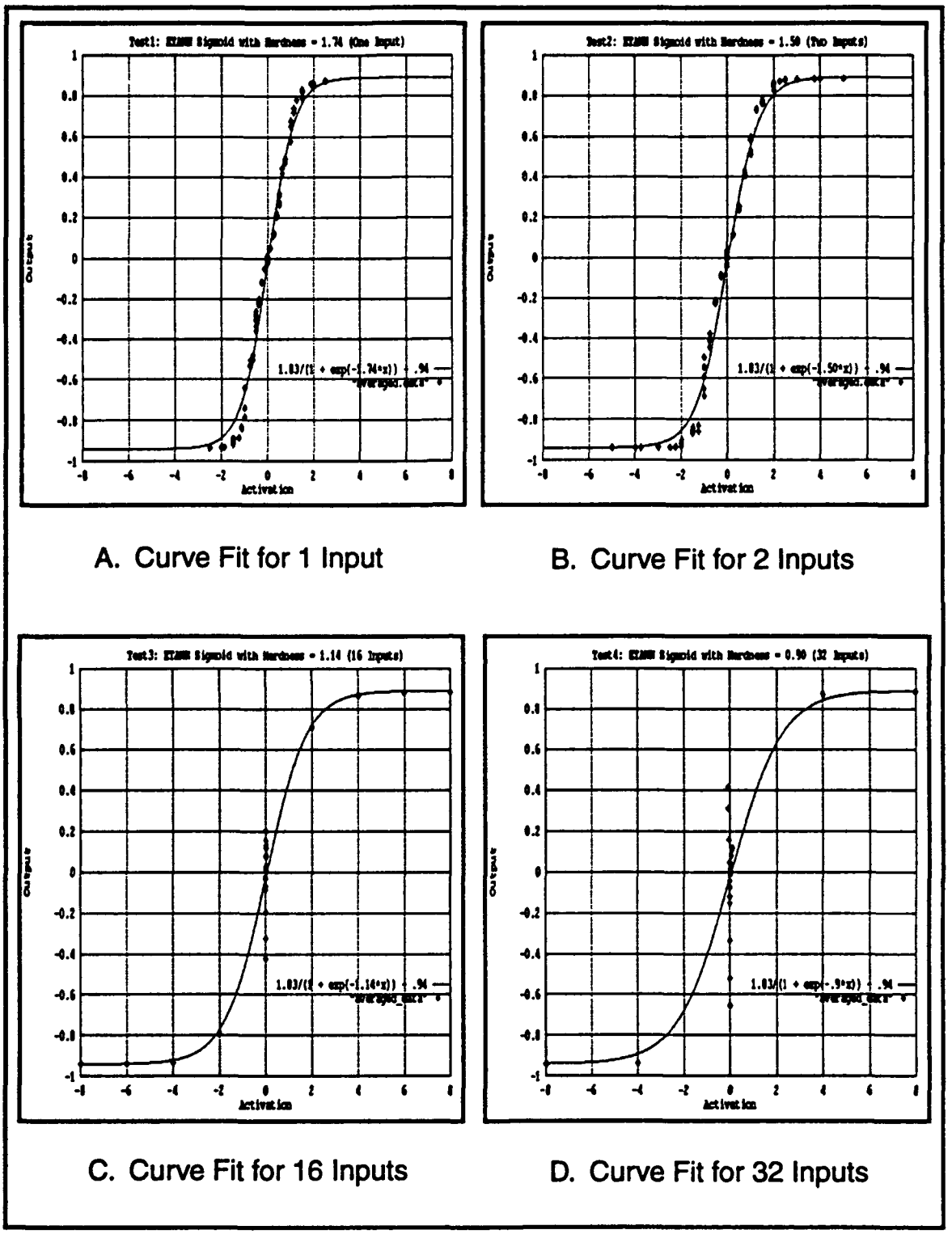

Figure 9. Curve Fit for Various Inputs

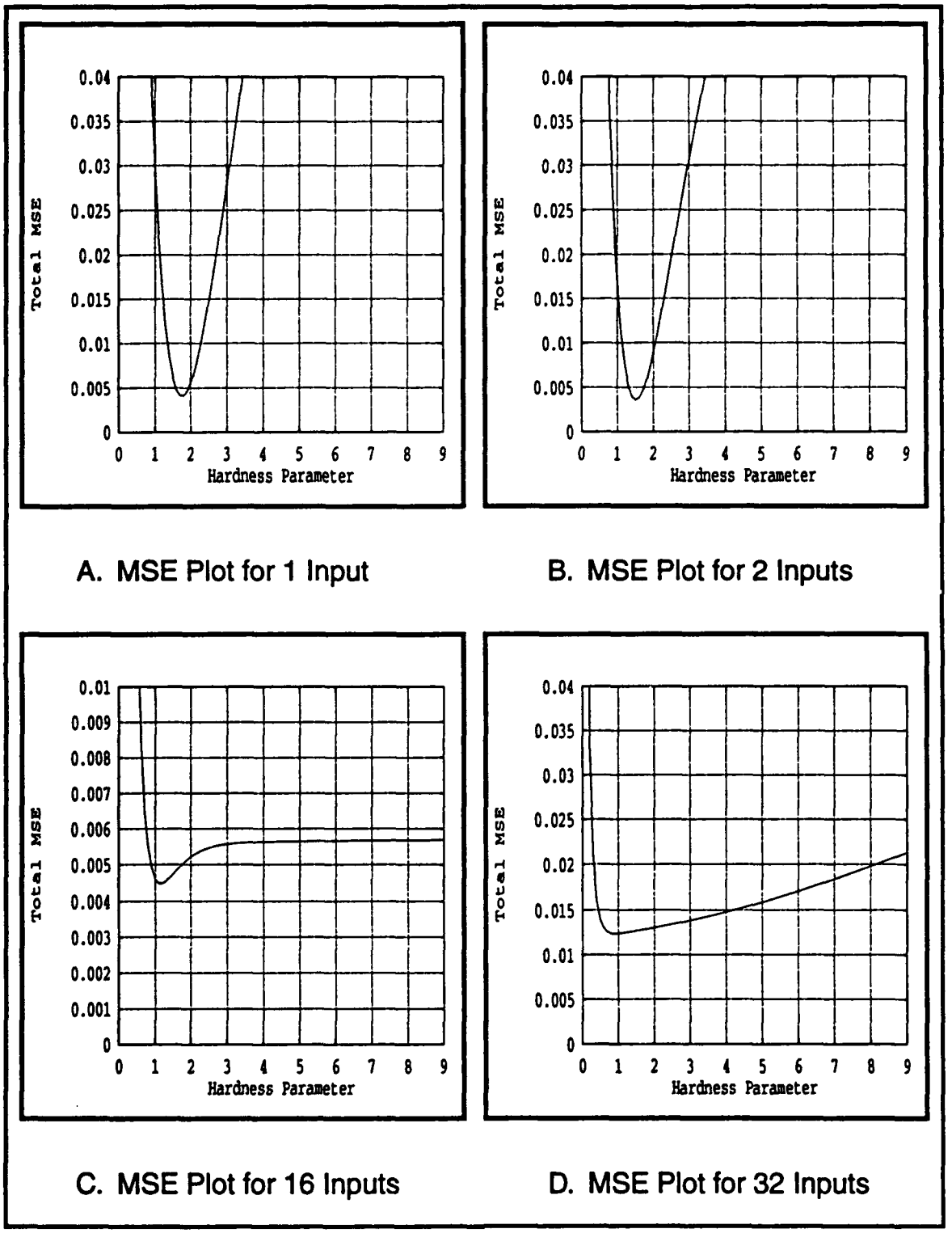

Figure 10. MSE Plots

following transfer function suffices to characterize the ETANN device for simulation.

$$Neuron\ Output = \frac{1.83}{1 + e^{-1.74 \sum (synaptic\ contributions)}} - .94 \qquad (5)$$

Note that the hardness parameter which provided the best results (lowest error in training the test network) was derived from test number one for a single neuron.

4.2.3 Simulator and ETANN Implementation Results for a 3-Class Problem

Simulation and implementation tests were performed to assess the accuracy of the ETANN sigmoidal transfer function characterized above. Tables 4 and 5 illustrate simulator and implementation results (averaged over three runs) while using iBrain-Maker (v2.02) and Neural Graphics as simulators. The Neural Graphics simulator was coded with the sigmoidal transfer function found in this research. The iBrain-Maker simulator, which is a commercial software system that comes with iNNTS (3), uses a look-up table rather than a coded sigmoid; using the high, low, and gain values that (3) provided in their transfer function menu, the look-up table (resolution unknown) probably emulates the following transfer function:

$$iBrainMaker\ Neuron\ Output = \frac{1.765}{1 + e^{-1.0 \sum (synaptic\ contributions)}} - .915 \qquad (6)$$

Note: The iDynaMind (v2.0) (which is also a part of iNNTS) was not included in the test because for large-class problems with thousands of training vectors (to be used in later experiments), the simulator's limitations are exceeded.

Comparative testing was done to check Neural Graphics' results with those of a working simulator, iBrainMaker. Both simulators trained on the same data and were also tested with the same test data; the initial weights were randomized for both simulators. After training, the weights were downloaded from each simulator to an ETANN device (reused the same device) for testing. Again, the same test data

Table 4. ETANN and iBrainMaker with 3-Classes

Number of Epochs	*Training Time(min:sec)*	*Simulator Training*	*Simulator Test*	*ETANN Test*
18	01:27	86%	82%	75%
37	02:53	88%	88%	78%
51	03:56	89%	92%	83%

was used on the device in each independent test without further training to check classification of test data. Three independent tests were performed to establish a classification average. The Neural Graphics simulator test results were matched by ETANN's test result. The iBrainMaker simulator performance, however, was less than ideal. The ETANN tests using iBrainMaker's trained weights were on average 15 percent lower than those tests using Neural Graphic's weights for the 3-class network. The set-up configuration for Neural Graphics and iBrainMaker was as follows:

- Training vectors: 540 (class 1 = 180; class 2 = 177; class 3 = 183)
- Test vectors: 60 (class 1 = 20; class 2 = 23; class 3 = 17)
- Features: 16 (Radar Signal Characteristics (5))
- Hidden nodes: 20
- Classes: 3 (number of emitters)
- Iterations: 540 iterations equals one epoch in this network
- Note: In Table 4 the time listed includes only training time.
- Note: In Table 5 the total training time plus testing time is listed. After each epoch of training, Neural Graphics automatically performs a network test using all test vectors.

Table 5. ETANN and Neural Graphics with 3-Classes

Number of Epochs	*Training Time(min:sec)*	*Simulator Training*	*Simulator Test*	*ETANN Test*
18	01:01	91%	93%	93%
37	02:10	95%	93%	93%
51	03:00	97%	95%	95%

4.2.4 Simulation and ETANN Implementation Results for an 8-Class Problem

Table 6 and 7 show the test results for an 8-class problem. These eight classes are a subset of the first eight classes of the 30-class radar emitter data. As clearly indicated from the data in Table 6, iBrainMaker had a difficult time training on the data. There is no known explanation as to why iBrainMaker performed so poorly; on the other hand, Neural Graphics did a superb job of training on the exact training and test data used in iBrainMaker.

Table 8 shows the Chip-In-Loop (CIL) training results with iBrainMaker on the 8-class problem. CIL training results were averaged over three separate training events, each totaling 1,842 epochs of training. The highest on-chip test result was 88 percent, which is 10 percent lower than the on-chip test result after training with Neural Graphics.

Because of time constraints, it was decided after this test that further training and testing off-chip would be limited to the Neural Graphics simulator even though there is no CIL capability at this time. The iBrainMaker simulator trains much too slowly for large class problems with a lot of data. In addition, it produces poor classification results from the radar emitter data used in this research, even though CIL training was used to improve on-chip classification performance.

Table 6. ETANN and iBrainMaker with 8-Classes
400 Training Vectors and 200 Test Vectors

Number of Epochs	*Training Time(hr:min:sec)*	*Network Size*	*Simulator Training*	*Simulator Test*	*ETANN Test*
1,843	2:33:00	16x16x8	64%	71%	72%
4,774	6:37:57	16x16x8	70%	72%	72%
2,045	4:37:19	16x20x8	80%	76%	77%

Table 7. ETANN and Neural Graphics with 8-Classes
400 Training Vectors and 200 Test Vectors

Number of Epochs	*Training Time(min:sec)*	*Network Size*	*Simulator Training*	*Simulator Test*	*ETANN Test*
100	01:40	16x16x8	95%	94%	98%

Table 8. CIL Results with iBrainMaker
400 Training Vectors and 200 Test Vectors

Simulator Epochs	*Network Size*	*Simulator Train/Test*	*CIL Epochs*	*CIL Train/Test*
1,842	16x16x8	73%/69%	0	70%/65%
"	"	"	10	88%/84%
"	"	"	20	89%/79%
"	"	"	30	89%/83%
"	"	"	40	90%/87%
"	"	"	50	90%/85%
"	"	"	60	92%/86%
"	"	"	70	92%/86%
"	"	"	80	94%/86%
"	"	"	90	90%/88%
"	"	"	100	88%/69%

Table 9. Training and Testing Results I.

Iterations (x1000)	*Classes*	*Training Vectors (Class/Total)*	*Test Vectors (Class/Total)*	*Total Error*	*Neural Graphics Simulator (Train/Test)*	*ETANN (Test Only)*
70	8	50/400	25/200	19	94%/91%	92%
61	8	75/600	25/200	24	94%/94%	92%
67	14	50/700	25/350	30	94%/93%	95%
84	14	75/1050	25/350	34	93%/93%	93%
74	20	50/1000	25/500	43	91%/89%	88%
86	20	75/1500	25/500	52	87%/85%	86%
78	26	50/1300	25/650	61	76%/76%	72%
98	26	75/1950	25/650	48	82%/81%	59%
100	30	50/1500	25/750	45	81%/80%	72%
100	30	75/2250	25/750	56	78%/78%	77%

4.3 Phase Two Research Findings

4.3.1 Single Chip Tests Table 9 is a summary of first-cut results while using Neural Graphics to simulate the ETANN device for a real application—solving the radar emitter identification problem. Note: All test results were averaged over three independent training/test runs. The single chip tests (as well as multi-chip tests) involved training using the normal back propagation algorithm (no adaptive measures or momentum). The best training result between 0 and 100,000 iterations was chosen—that is the reason for the difference in iteration numbers in Table 9. The number of training and test vectors were chosen to correspond to the baseline results listed in Chapter 3. As the number of classes increase, the ETANN test results get worse (in general). The chip resolution is limited at about three decimal places of accuracy, whereas the simulator trains/tests with approximately six decimal places of accuracy. Although the results so far do not come close to matching baseline results listed in Chapter 3, these results are a starting point to improve upon.

Several training improvement techniques were tried in order to get the best training and testing results. First of all, the weights in the Neural Graphics simu-

lator were growing steadily as training iterations increased. The ETANN device is constrained to accept weight values between $-2.5 \leq w_{ij} \leq 2.5$. The first approach to solve the problem was to scale the weights after training and write them to the ETANN device. An experiment involving real training data was performed as an attempt to solve the problem. Failure was the result. Apparently a lot of information is lost when scaling the weights; thus, poor testing performance can be expected.

The next test involved changing the Neural Graphics weight update code. In order to keep the weights within range of the ETANN device, the Neural Graphics weight update program routine was modified to limit the weight values. The modified program routine is in Appendix F. The biases, on the other hand, are allowed to exceed the range, $-2.5 \leq w_{ij} \leq 2.5$, because the ETANN device allows the programmer to use up to seven biases per neuron. Although Neural Graphics trains with one bias per neuron, the bias value may be divided by any integer between and including one and seven. For this research, six was the deviser used, and the result was written to six biases. Consequently, the bias values in Neural Graphics were limited to a value of 15.

To improve network performance and lower the total error, the Neural Graphics code was modified. Recall the ETANN sigmoidal transfer function experiment results. Based on test results, it was concluded that the ETANN device incurs more error when training to $+1$ for the desired class and to -1 for the non-desired classes; consequently, if the actual device high and low threshold values were assigned as the desired class and non-desired classes, respectively, then total error could be reduced in training. In simulator tests performed for this research, training results did improve and the total error dropped significantly when training to the modified desired and non-desired levels (compare training and testing results I and II). The ETANN levels were characterized earlier in this chapter as 0.89 (high) and -0.94 (low) and were coded in the Neural Graphics simulator for training/testing improvement.

Further improvements in training were made by using the exact derivative calculated from the sigmoidal transfer function, denoted by f, derived from the ETANN. The derivative equation is the following:

$$\frac{\partial f(1.74\alpha)}{\partial \alpha} = \frac{\partial\{-1.83(1+e^{-1.74\alpha})^{-1} - .94\}}{\partial \alpha} \quad (7)$$

$$= .95[.94 + f(1.74\alpha)][.89 - f(1.74\alpha)] \quad (8)$$

The result of using this derivative function as opposed to the precise derivative of a tanh function (see Appendix A) was increased training speed and accuracy while using Neural Graphics—see Table 10. Although not perfect results, this table shows a definite improvement over Table 9. As the number of classes increased in the data sets, classification accuracy decreased as expected with few exceptions. Generally, as the number of training vectors are increased, the number of *required* training epochs increases; this is why there were some exceptions to increasing classes and decreasing classification accuracy. The ETANN classification results generally followed the same trend as the simulator's, except for the two 30-class networks, which were the trend breakers. Resolution breakdown is a probable explanation for this experiment outcome. The ETANN's resolution is about 7.5 bits for the weight values, whereas the Neural Graphics simulator has more than 32 bits resolution. There will be a margin of error regardless of what simulator a person uses, but the goal is to minimize that error, which is one of the main objectives of this research. Fine tuning can always be done through chip-in-loop training, which is the final step towards realizing a fully trained network. Chip variation and resolution can usually be compensated for by the chip-in-loop training method, but poor training and testing results cannot.

According to (28), if enough hidden neurons are used in a two layer network (input$\times$ hidden $\times$ output), most networks can be accurately classified (given good features and separable data). An experiment was performed to find an efficient num-

Table 10. Training and Testing Results II.

Iterations (x1000)	*Classes*	*Training Vectors (Class/Total)*	*Test Vectors (Class/Total)*	*Total Error*	*Neural Graphics Simulator (Train/Test)*	*ETANN (Test Only)*
40	8	50/400	25/200	11	95%/94%	98%
40	8	75/600	25/200	15	94%/96%	97%
40	14	50/700	25/350	15	96%/94%	94%
40	14	75/1050	25/350	20	95%/95%	96%
40	20	50/1000	25/500	20	95%/93%	91%
40	20	75/1500	25/500	29	92%/92%	92%
40	26	50/1300	25/650	29	90%/89%	92%
40	26	75/1950	25/650	37	90%/89%	65%
40	30	50/1500	25/750	38	88%/88%	71%
40	30	75/2250	25/750	48	88%/89%	80%

ber of hidden neurons that would give good classification results for the radar emitter features. The data was trained using the standard back propagation algorithm. Table 11 shows that generally as the number of hidden neurons is increased the total error goes down, and the classification accuracy goes up. The principal drawback to adding more hidden neurons is slower training speed. By observation, the ETANN chip appears to have some difficulty in achieving better than 83 percent classification for the 30-class network. A difference in chip resolution versus simulator resolution is a likely reason for the discrepancy. In experiments with fewer than 26 classes, the ETANN performed well for this set of data. Classification results will always be problem dependent—the number of training and test vectors, the separability of classes, and the quality of features used are all important considerations.

Table 12 shows the final training and testing results for a single chip implementation of the radar emitter feature data. What perhaps is not obvious is that as the number of classes increase, so does the total error. As the number of training epochs increase, the total error decreases. The ETANN test results worsen as the number of classes increase for this particular problem. This trend suggests the ETANN does

Table 11. Training and Testing Results III.
75 Training Vectors Per Class (2250 Total)
and 25 Test Vectors Per Class (750 Total)

Iterations (x1000)	*Classes*	*Network Size (input-hidden-output)*	*Total Error*	*Neural Graphics Simulator (Train/Test)*	*ETANN (Test Only)*
200	30	16x16x30	48	82%/81%	76%
200	30	16x20x30	44	86%/85%	81%
200	30	16x25x30	42	92%/90%	77%
200	30	16x30x30	41	91%/91%	72%
200	30	16x35x30	41	92%/92%	83%

not have enough bit resolution to separate all the classes with the simulator trained weights. The classification results, as with all results in this chapter, are an average of three measurements.

Perhaps the problem cannot be solved to user satisfaction using a single chip with weight resolution of 7–8 bits without resorting to chip-in-loop (CIL) training. The primary reason for having characterized the ETANN sigmoidal transfer function in the first place was to limit the amount of CIL training required, because CIL training decreases the useful life of the ETANN (3:19-8). Due to limited time, a CIL program was not written, but it is recommended for future research. With considerable effort, it is possible to code a C routine to perform CIL training. After 10 to 20 epochs of CIL training, classification accuracy should be improved. CIL compensates for chip variance and loss of resolution when going from simulator to chip.

4.3.2 Multi-Chip Testing Using the EMB The ETANN Multi-chip Board (EMB) is a platform that allows a designer to implement up to eight chips in various configurations for neural computing. The EMB was used in this research to implement the primary thesis problem solved in simulation (5). The best training

Table 12. Training and Testing Results IV.
75 Training Vectors/Class (2250 Total)
and 25 Test Vectors Per Class (750 Total)

Number of Epochs	*Total Error*	*Network Size*	*Simulator Training*	*Simulator Test*	*ETANN Test*
44	25	16x35x30	95%	93%	75%
89	24	16x35x30	96%	94%	83%
133	24	16x35x30	96%	95%	81%
177	24	16x35x30	96%	95%	82%

results achieved in (5) came from using a hierarchical system of classifying 30 radar emitters. The hierarchical approach which gave the best test results in simulation is implemented in ETANN hardware using three devices on the EMB platform for performance evaluation and simulator comparison. First, a brief description of the hierarchical approach is presented, and then the hardware results are shown.

4.3.2.1 Hierarchical Approach As described in (5), the hierarchical approach divides a large-class network into smaller networks with one or more grouping networks. For example, if one were to divide a 30-class problem into two subnetworks, a total of three networks would be needed to perform the classification task. Suppose the networks for the hierarchy used in research are listed as follows:

1. A 2-class network for separating the 30 classes into two groups—used as a switch to control the other two networks.
2. A 12-class network (bottom 12) to classify group one.
3. An 18-class network (top 18) to classify group two.

This hierarchy is but only one example of the infinite number of possible designs. It just so happens that this design is the network hierarchy used in this research. Figure 11 shows a block diagram for the hierarchical approach used in this research.

Network two acts as a switch on the output that selects one of two classes; one class contains all the bottom 12 classes, and the other contains all the upper 18 classes.

4.3.2.2 Hierarchical Configuration Results Until now most of the previous experiments were performed on a single chip, discounting chip-to-chip differences. In the following experimental results, chip-to-chip variances were investigated for a multi-chip implementation. Six chips were used in the experiments. On-chip and chip-to-chip variance test results are in Tables 13 and 14.

Table 13 shows the on-chip variance test results for six chips in three different networks. The networks listed in this table are one example of a hierarchical network used to solve the radar emitter problem. To compute a standard deviation, three ETANN tests were performed on every chip, and the result was averaged. Variation on a single chip is practically negligible.

On the other hand, chip-to-chip variation as shown in Table 14 is more significant, averaging about ± 3 percent for all the networks shown. This finding reinforces the need for chip-in-loop training, which was not performed in this research because of limited time. The data used to train the hierarchical networks were partitioned as follows: 1) Network One: 2250 training vectors (18/12 split into two classes), 2) Network Two: 900 training vectors (bottom 12 classes) , and 3) Network Three: 1350 training vectors (top 18 classes). The number of training iterations for each network was as follows: 1) Network One: 600k, 2) Network Two: 500k, and 3) Network Three: 400k.

Table 15 shows the final hierarchical implementation results for three different network sizes. The hierarchical probability was computed using the actual results from on-chip tests. The simulator tests for the hierarchical network are the same as those listed in Table 14.

Without chip-in-loop training, it appears that the best an ETANN device can do with this particular data (30-classes) is about 87 percent classification rate. In-

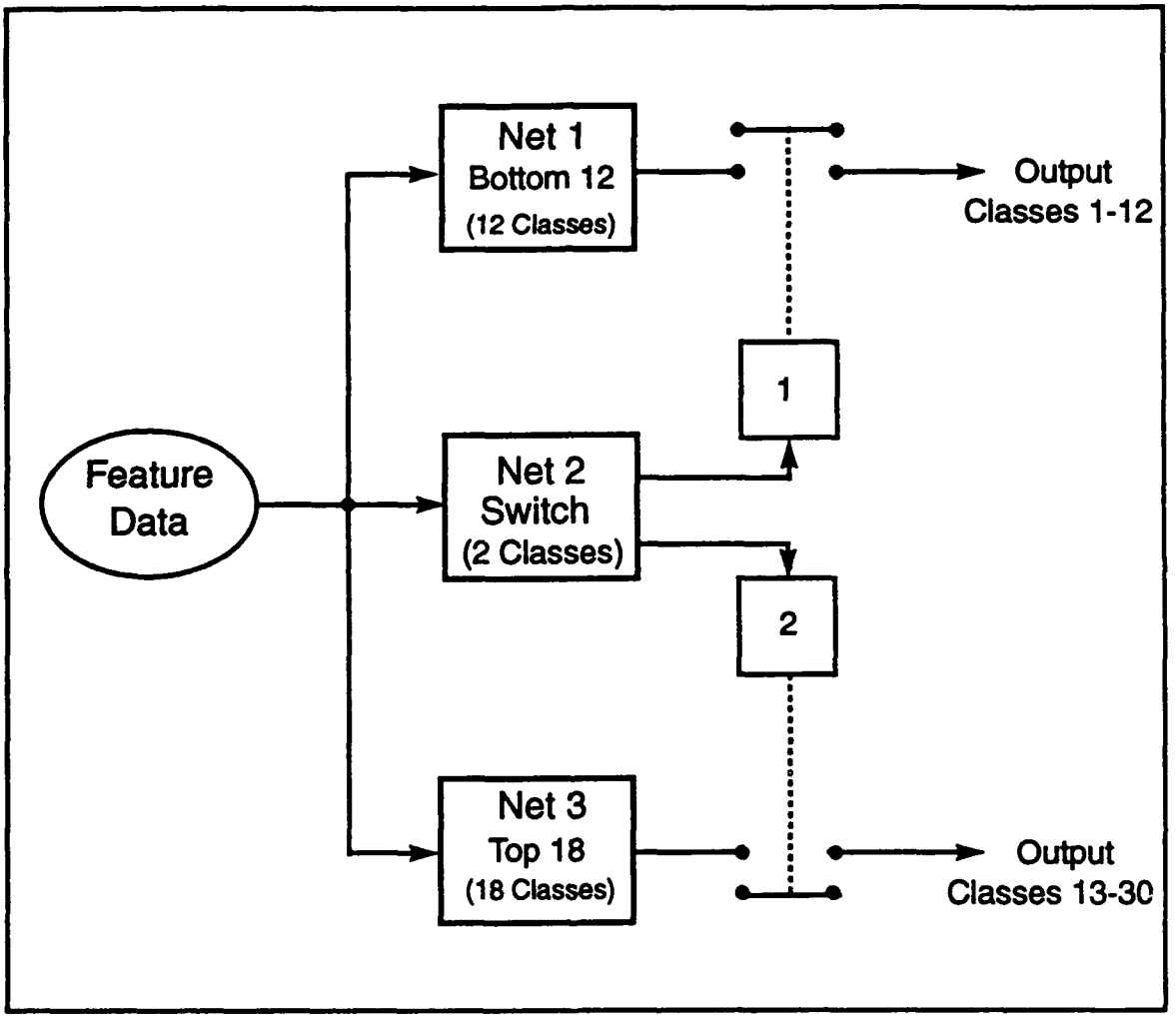

Figure 11. Block Diagram for Hierarchical System

Table 13. On-Chip Variance Test Results

Chip	*Network Size*	*Simulator Train/Test*	*Total Error*	*ETANN Test Avg.*	*Standard Deviation*
1	16x20x2	99%/96%	23	95%	±.1%
2	"	"	"	86%	±.2%
3	"	"	"	90%	±.1%
4	"	"	"	84%	±.1%
5	"	"	"	87%	±.1%
6	"	"	"	86%	±.1%
1	16x20x12	97%/96%	18	95%	±.0%
2	"	"	"	94%	±.0%
3	"	"	"	95%	±.2%
4	"	"	"	96%	±.4%
5	"	"	"	91%	±.0%
6	"	"	"	96%	±.2%
1	16x20x18	99%/98%	19	97%	±.1%
2	"	"	"	82%	±.2%
3	"	"	"	89%	±.1%
4	"	"	"	83%	±.1%
5	"	"	"	89%	±.3%
6	"	"	"	88%	±.0%

Table 14. Chip-to-Chip Variance Test Results

Network Size	*Simulator Test*	*Test Avg. of 6 Chips*	*Combined Deviation*
Net1 16x20x2	93%	90%	±4.3%
Net2 16x20x12	96%	96%	±1.6%
Net3 16x20x18	98%	96%	±3.6%
Net1 16x25x2	93%	89%	±1.9%
Net2 16x25x12	96%	92%	±2.8%
Net3 16x25x18	98%	95%	±2.9%
Net1 16x30x2	96%	93%	±1.3%
Net2 16x30x12	96%	94%	±3.6%
Net3 16x30x18	98%	92%	±4.0%

creasing the number of hidden nodes did not improve performance for the hierarchical networks as it did in single chip implementations (assuming the same number of training iterations), but it did decrease the amount of deviation from the average result. More training might cause a network with more hidden neurons to improve classification accuracy, or it might cause the network to generalize poorly. But from observation of the weight file, the majority of the weights were near their maximum values that they could take on (an indication that the network was near "brain dead").

The term "brain dead" is used to describe a network that will not train any longer. If the network is near some acceptable range of classification, then chip-in-loop (on-chip) training should be all that is needed to train the network to some acceptable error. If the network is "brain dead" and the network is still performing unacceptably, then try restarting training with a new set of random weights.

The drawbacks to using a hierarchical approach are added complexity, additional number of devices, and increased processing time (about 3μsec for this hierarchy). Furthermore, there are no guarantees of better performance over a single chip implementation. Even though in this training example the on-chip test results were better for the hierarchical networks than for the single chip networks, the single chip networks performed better in simulation (compare simulation results in the tables). Because of resolution loss when going from simulator to chip with higher-order networks, on-chip tests will be poorer for single chip implementations than for hierarchical (multi-chip) implementations of the same network problem.

4.3.2.3 Research Sub-Objective Findings The primary objective of this research was to implement an artificial neural network in the ETANN hardware to classify radar emitter data. The sub-objectives in this research include the following:

1. Minimize the processing time and maximize the classification accuracy.
2. Develop a technique to load training weights from Neural Graphics to iNNTS.

Table 15. Hierarchical Implementation Results

Hierarchical Network Size Architecture	*Hierarchical Probability*	*Combined Deviation*
16x20x2 / 16x20x12 / 16x20x18	87%	±9.5%
16x25x2 / 16x25x12 / 16x25x18	84%	±7.5%
16x30x2 / 16x30x12 / 16x30x18	86%	±8.9%

3. Estimate processing time to make a classification in hardware.
4. Compare resolution differences between the hardware device and simulator
5. Calculate an implementation loss factor.

To minimize processing time yet maximize classification accuracy requires a designer to consider trade-offs. The best on-chip classification accuracy came from using a hierarchical implementation; however, the fastest network was a single chip implementation using a two layer (inputs x hidden x outputs) network.

A method of transferring the Neural Graphics trained weights to iNNTS was accomplished with success. Appendix H contains the C tools necessary to accomplish this objective.

Processing time is dependent on network architecture. This means as more layers are embedded in the network, the longer the time it will take to process the feature information and produce an output. Each layer of a network takes 3 μsec to process input data and produce an output. For the single chip implementation to the thesis problem (assuming two layers), the processing time takes 6 μsec. For the hierarchical network in this research, however, the processing time takes a minimum of 9 μsec—6 μsec for three independent networks (on separate chips) to process the feature data simultaneously and 3 μsec to select the winning output.

According to (20), chip weight resolution is about 7.5 bits (about three decimal places), and device processing resolution is about 6.5 bits (between two and three

decimal places). The Neural Graphics simulator uses more than 32 bits of resolution in processing numerical data (six decimal places). The trained weights from Neural Graphics were rounded off at eight bits resolution (three decimal places). For the data used in this research, the ETANN chip accuracy began dropping off with eight-class networks and was severely degraded with 26-class networks (in early experiments). After using improved training methods (in section 4.3), accuracy degradation was reduced to more acceptable levels for up to thirty-class networks—fair results considering that no chip-in-loop training was used to compensate for chip-to-chip variance or loss of resolution.

The loss of resolution when going from simulator to chip is network dependent, but was more pronounced in single chip implementations than in hierarchical networks for the same network problem. Hierarchical networks lower the loss of resolution by breaking large-class problems into sub-networks with fewer classes. The fewer the number of classes, the less the loss of resolution when going from simulator to chip. The reason is because of network complexity. It takes less weight resolution to separate fewer classes, but as the amount of training data increases, so does the required weight resolution.

The implementation loss computed for this research assumed no chip-in-loop (on-chip) training. For a single chip network implementation (16 x 35 x 30), the average (of four networks) implementation loss per chip was 14 percent. For the hierarchical network, the average (of nine networks) implementation loss per chip was 3 percent, but the hierarchy consists of three chips—the total average implementation loss equaled 9 percent. Again, implementation loss is a factor of network size, data complexity, amount of data, and chip variance. Loss factors will be different for other network implementations.

4.3.3 Feature Saliency The Neural Graphics program provides a measure of the importance of a particular feature in the network classification process, and it is called saliency (5:29).

Cameron used five data sets in 25 runs to determine the saliency of each feature. He then calculated average and final ranking of each feature over all the runs—see Table 16.

- Classes: 30
- Training vectors: 50 per class
- Test vectors: 25 per class
- Hidden nodes: 20
- Iterations: variable; 10,000 to 30,000
- Special note: Neural Graphics toggled *Saliency On*

The most important feature for the classification process was feature 16; however, there was not a significant drop in network accuracy when feature 16 was omitted. This result could be due to the redundancy of information contained in the other 15 features. The other features were less important; there were no large discontinuities in the ranking (5:30).

4.4 Conclusion

This chapter presented findings on ETANN sigmoidal transfer function tests and on single chip and multi-chip experiments. Test results showed that the ETANN's sigmoidal transfer function can be accurately characterized through simple experiments.

The thesis problem and sub-objectives were met with little difficulty once the Neural Graphics simulator was successfully modified to emulate the ETANN device

Table 16. Feature Saliency Ranking (5:30)

Feature	Avg. Rank	Ranking
1	4.4	2
2	11.2	12
3	12.6	13
4	10.5	11
5	8.3	9
6	7.1	6
7	7.5	8
8	5.6	5
9	5.0	4
10	7.2	7
11	12.8	14
12	14.0	15
13	14.2	16
14	10.1	10
15	4.5	3
16	1.0	1

and a method of porting the weights out of Neural Graphics into iNNTS was developed. The maximum on-chip classification accuracy for a single chip implementation of the 30-class problem without chip-in-loop training was 83 percent, using 35 hidden neurons. Again without chip-in-loop training, the maximum on-chip classification accuracy for a hierarchical configuration with the 30-class problem was 87 percent. The total average implementation loss was found to be 9 percent for the hierarchical network.

From a feature saliency experiment, the most important feature found in the classification process was feature 16, and the least important was feature 13. For security reasons, the features of the data are only known as radar emitter characteristics.

V. CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusions

Results of this research are ground-breaking in several categories. First of all, no radar emitter classification problem has been implemented in ETANN hardware until now. Also, there have been no reports of using customized simulators on workstations to train networks for the ETANN chip. Furthermore, there have been no published reports of the ETANN chip's sigmoidal transfer function being characterized for non-binary operation. Below is a summary of the significant contributions in this research:

1. Characterized the ETANN sigmoidal transfer function. The ETANN sigmoidal transfer function was not previously characterized; now it has been accurately characterized and tested in a custom simulator, Neural Graphics.
2. Customized the Neural Graphics simulator for ETANN emulation. Neural Graphics was developed primarily for emulating the Cybenko type of network using $1/(1+e^{-\alpha})$ as a sigmoid function. Now, Neural Graphics can emulate the ETANN chip using a modified tanh function, which was derived during experiments on the ETANN chip.
3. Developed interface software with iNNTS. At the beginning of this research there was no software to interface with iNNTS for the purpose of writing weight values, which were trained on a custom simulator, to the ETANN chip. In addition, there was no software for testing the on-chip classification accuracy. Four C programs were written to interface with iNNTS—two programs each for the single chip and the ETANN Multi-chip Board (EMB) modes of operation.
4. Implemented thesis problem in single chip and multi-chip configurations. Until now the 30-class radar emitter identification problem had only been solved in

simulation using a Cybenko type sigmoidal transfer function on Neural Graphics (5).

5. Defined feature saliency for thesis problem. Feature saliency was defined previously in (5) and was accomplished a second time during this thesis research. The results are the same.

For the purpose of emulating an ETANN device in a custom simulator, a sigmoidal transfer function was derived, tested, and implemented as a model for $V_{GAIN} = 3.3V$, 64-input operation. The result is the following equation:

$$Neuron\ Output = \frac{1.83}{1 + e^{-1.74 \sum (synaptic\ contributions)}} - .94 \tag{9}$$

Neural Graphics, an AFIT neural network simulator running on a Silicon Graphics (model 4D) workstation, was customized for use with the ETANN chip. Customizing Neural Graphics required changing the coded sigmoidal transfer function, coding a limitation routine on weight values to keep weight values below the ETANN's maximum value, adding more bias weight values for chip emulation, and modifying Neural Graphics' "makeinput.c" file to optimize desired and non-desired training values in the training (i.e., in general a simulator coded with the tanh transfer function should train to -1 for non-desired class and +1 for desired class).

In addition to modifying the sigmoidal transfer function, there were several other ETANN emulation problems to overcome in Neural Graphics. For instance, the weight values in Neural Graphics grew steadily as the number of training iterations increased, and thus they exceeded the ETANN chip's dynamic range in the course of training. Moreover, the unmodified version of Neural Graphics trains a network using a single bias weight value per neuron, whereas the ETANN chip may use up to seven bias weight values per neuron. The additional neurons give added dynamic range to the weight dimension space, which makes simulator and chip training easier. In

another problem the Neural Graphics weight values were not written in a compatible format for programming in iNNTS. The solution to this problem required developing software to interface the Neural Graphics' weight file with iNNTS and writing the weight values to a chip, both in single-chip and ETANN multi-chip board (EMB) modes of operation. Furthermore, a classification program was written to classify test vectors on the ETANN chip. All of these problems were solved during the first phase of research (covered in Chapter 4).

In the second phase of research, emphasis was on solving the radar emitter identification problem and the sub-objectives. The maximum on-chip classification accuracy for a single chip implementation without chip-in-loop training was 83 percent, using 35 hidden neurons (16 × 35 × 30). Again without chip-in-loop training, the maximum on-chip classification accuracy for a hierarchical configuration with the 30-class problem was 87 percent, using 20 hidden neurons (chip one [16 × 20 × 2], chip two [16 × 20 × 12], and chip three [16 × 20 × 12]).

Fast operation (speed), in general, is a major consideration for emitter classification in hardware. The highest on-chip classification accuracy from these experiments came from using a hierarchical implementation; however, the fastest network came from using a single chip implementation of two layers (input × hidden × output). The speed versus classification accuracy trade-off is a common factor in simulation as well as in hardware implementation.

A method of transferring Neural Graphics' trained weights to iNNTS was accomplished by formatting the weight values in an ASCII file and then running an interface program with iNNTS to write the weights to a chip. Developing this method was essential to implementing a neural network in hardware using the customized Neural Graphics simulator.

Implementation loss factors are network and data dependent. For the hierarchical network investigated in this research, the total average implementation loss (from three experiments) was found to be 9 percent. In other words, one can ex-

pect to lose classification accuracy by approximately 9 percent when implementing a device with the Neural Graphics simulator trained weights, without chip-in-loop (on-chip) training, for this particular data.

The last experiment was to determine feature saliency. Chapter 4 shows the saliency results with the features in rank order. The most important feature for the classification process was feature 16, and the least important was feature 13. For security reasons, the features of the data are only known as radar emitter characteristics.

5.2 Recommendations

Topics in this thesis could be explored further. The single most important topic to continue research is on development of a Chip-In-Loop (CIL) training program using the iNNTS software interface. With a CIL capability, one can finish training the simulator-trained chips (as done in this research) to an acceptable error rate.

Another topic to continue research on is the use of different criteria for the partitioning of the classes for hierarchical systems. Perhaps one could yield further increases in classification accuracy. If both of these recommendations are followed, increases in data and class size can be investigated and the results compared to those in this thesis.

5.3 Summary

This research has demonstrated that artificial neural networks can be used in the design of radar emitter identification systems and be implemented in hardware with acceptable accuracy (considering no chip-in-loop training) for 30 classes. It has also been demonstrated that the hierarchical approach to the 30-class problem results in a greater on-chip accuracy than that of a single chip implementation. The maximum on-chip classification accuracy for a single chip implementation was 83 percent. For a multi-chip implementation using a hierarchical approach, the maxi-

mum on-chip classification accuracy was 87 percent. The drawbacks to a hierarchical approach, however, are added complexity, added number of devices, and added processing time (about 3μsec for this hierarchy). For the hierarchical network, the total average implementation loss was found to be 9 percent in this problem. It is recommended that a chip-in-loop training interface with iNNTS be developed in future research efforts. And finally, investigation into the use of different criteria for partitioning the classes for hierarchical systems could yield further increases in classification accuracy. Thus, following these recommendations would allow further research efforts in ETANN hardware implementation with larger class problems than those studied here.

Appendix A. *Partial Derivative of ETANN Sigmoidal Transfer Function*

This appendix presents a derivation for the partial derivative of the general type of sigmoid function used in the ETANN chip. The vanilla back propagation algorithm as described by Rogers and others (28) uses a unipolar sigmoid function, sometimes referred to as a squashing or logistic function. Below is a derivation of the partial derivative for the general sigmoid function used in the ETANN device, sometimes referred to as a bipolar or symmetric sigmoid.

$$
\begin{aligned}
f(\alpha) &= [\frac{2}{(1+e^{-\alpha})} - 1] \\
where\ \alpha &= \sum_{i=1}^{n+1} \omega_i x_i + \theta \\
assuming\ a\ gain\ coefficient &= 1.0 \\
\theta &= total\ bias\ on\ a\ neuron \\
\frac{\partial f(\alpha)}{\partial \alpha} &= -2(1+e^{-\alpha})^{-2} \cdot \frac{\partial e^{-\alpha}}{\partial \alpha} \\
&= -2(1+e^{-\alpha})^{-2} \cdot e^{-\alpha} \cdot (-1) \\
&= \frac{2e^{-\alpha}}{(1+e^{-\alpha})^2} \\
&= [\frac{2}{(1+e^{-\alpha})} - 1 + 1][\frac{2e^{-\alpha}}{(1+e^{-\alpha})}] \cdot \frac{1}{2} \\
&= \frac{1}{2}[1+f(\alpha)][\frac{2e^{-\alpha}+2-2}{(1+e^{-\alpha})}] \\
&= \frac{1}{2}[1+f(\alpha)][2 - \frac{2}{(1+e^{-\alpha})}] \\
&= \frac{1}{2}[1+f(\alpha)]\{1 - [\frac{2}{(1+e^{-\alpha})} - 1]\} \\
&= \frac{1}{2}[1+f(\alpha)][1-f(\alpha)] \\
&= \frac{1}{2}[1-f(\alpha)^2]
\end{aligned}
$$

For the ETANN sigmoidal transfer function defined in Chapter 4, Equation 5, it can be easily shown that the derivative results in the following equation:

$$\begin{aligned}\frac{\partial f(1.74\alpha)}{\partial \alpha} &= \frac{\partial\{-1.83(1+e^{-1.74\alpha})^{-1}-.94\}}{\partial \alpha} \\ &= .95[.94+f(1.74\alpha)][.89-f(1.74\alpha)]\end{aligned}$$

Recall that for backward error propagation the weights are updated as follows:

$$w_{ij}^{+} = w_{ij}^{c} - \eta\delta_j x_i + \alpha(w_{ij}^{c} - w_{ij}^{-})$$

where w_{ij} is the weight from node i to node j in the next layer, x_i is the output of node i, and δ_j is the error associated with node j. The η and α represent learning rates. The new weight is represented by w_{ij}^{+}, the current weight by w_{ij}^{c}, and the old weight by w_{ij}^{-}. Thresholds are adapted similarly where x_i is replaced by $+1$ if the threshold is added to the weighted sum and -1 if it is subtracted.

The δ_j for the ETANN equations are defined in this research as follows:

$$\delta_j = \begin{cases} .95[.94+y_j][.89-y_j](d_j-y_j) & \text{for output node j} \\ .95[.94+x_j][.89-x_j]\sum_k \delta_k w_{jk} & \text{for hidden node j} \end{cases}$$

where x_j and $y_j = f(1.74\alpha)$ in the hidden and output nodes respectively; d_j is the desired output for output node j, and y_j is the actual output. For the hidden nodes the δ_k are the errors for the layers above (28:55).

Appendix B. *How to Make an iBrainmaker Weight File*

B.1 Weight File Format

The file format for the weights file is best shown through an example. Figure 12 shows an example of a network with weighted interconnections between input and hidden neurons (nodes) and hidden and output neurons (nodes). This example illustrates only one hidden layer; there may be any number of hidden layers, but it has been shown that one hidden layer suffices (with enough hidden nodes) to characterize any function (28:52). The number of hidden nodes, however, is problem dependent.

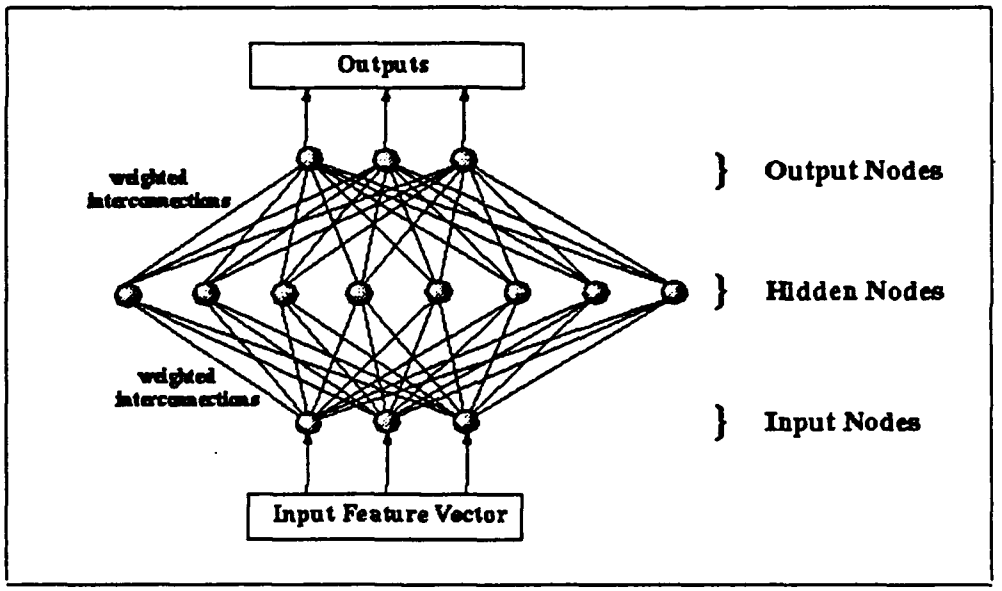

Figure 12. Network Example

The format for writing the weights to a file is shown below. Note that w_11 represents the weight from input node number one to hidden node number one. The weights are written from layer one (input to hidden layer) to the output layer. The file looks something like the following:

```
weights
w_11  w_21  bias  (weights to hidden neuron #1)
w_12  w_22  bias  (weights to hidden neuron #2)
.
.
.
w_14  w_24  bias  (weights to hidden neuron #4)
(empty line between layers)
w_11  w_21  w_31  w_41  bias  (weights to output neuron #1)
w_12  w_22  w_32  w_42  bias  (weights to output neuron #2)
```

Weights take on a range of -2.5 to 2.5 for ETANN configuration. Input values are limited in the range of -1.0 to +1.0. The number of bias weights per neuron is user defined (usually one to seven bias weights per neuron are used on the ETANN chip).

B.2 Writing Weights in Neural Graphics

Neural graphics can be modified to write your weights into a file of any desired format by simply modifying the "saver.c" and "general.c" files. An example of modifying "saver.c" and "general.c" is shown below. These files were modified to write the weights into a file named "weights.calvin" in the same format as iBrainMaker.

```
/***********************************
*   saver.c modification
***********************************/
/* section added to saver.c file */
write_calvin_weights(filename)
char filename[];
{
FILE *fpx;
int i,j,k;
fpx = fopen(filename,"w");
  if (fpx == NULL){
     printf("Couldn't open weight file ");
     ggexit();
  }
```

```
   fprintf(fpx,"weights \n ");
   loopi (parameters.layers){
         int sizex;
         loopj (net->layer1[i]->size_output){
               loopk(net->layer1[i]->size_input)
               fprintf(fpx,"%f ",net->layer1[i]->weights[j][k]);
               fprintf(fpx,"  %f",net->layer1[i]->theta[j]);
               fline;
         }
   fline;
   }
   fclose(fpx);
}
/*****************************************************
*              general.c modification
*****************************************************/
for(avg=1;avg <=avgs ;avg++){
   check_parameter_line(argc,argv);
   DO_NETWORK();
   hold_one_out();
   file_saliency(0);
   do_avg();
/* following line added to code in general.c file */
   write_calvin_weights("weights.calvin");
   write_weights("weights.temp");
   #ifdef TERMINAL
       endwin();
   #endif
}
```

Appendix C. *How to Design a Network for the ETANN Chip*

C.1 Overview of Design Process

Design of an ANN is basically a seven-step process of working through a logical description of the problem to be solved. First of all, decide whether the ANN is to predict, generalize, or recognize (34). Some examples include stock market average prediction, data generalization for loan risk analysis, and automatic target recognition systems using multi-sensor input to recognize tanks, trucks, and jeeps.

Next, decide what information will make good features for predictions, generalizations, or recognitions. Predictions are based on past information of pertinent data. For example, in predicting stock averages, information to be used for prediction might include previous stock prices, market interest rates, state of the economy, and world market averages, to name a few. Good features make good classifiers (predictors). The selection of information to use as features is more an art than a science. Follow your logical intuition and experiment with various types of data.

Following this last step, you will need to get lots of data. Generally, the more data, the better the predictor, generalizer, or recognizer. One rule states that using a minimum of three times the number of input features will give a rough estimate of how many training vectors are needed per class (Foley's rule) (28:60). If the total number of training vectors is less than twice the number of features, the training data error rate will be near zero regardless of the distribution, which is a result of Cover's theorem on capacity (28:60).

When building a feed-forward network, there are two things needed to make the network. One needs samples of input data and to know the desired results. The next step is to build the network. Two things are needed to make a network: a network definition and a collection of data. A network definition consists of assigning a number of input nodes to a number of hidden nodes to a number of output nodes,

see Figure 13. The number of inputs in Figure 13 is two, the number of hidden neuron nodes is two, and the number of outputs is one. The assignment of input nodes and output nodes is problem dependent. The assignment of hidden nodes is problem and data dependent. If the problem is linearly separable, then hidden nodes are not necessary, unless there is an enormous number of training facts. In that case add a hidden layer to improve network performance. Assign the number of hidden nodes to correspond to the network complexity (linearly separable versus non-linearly separable) by trial and error results. The goal is to train effectively with as few hidden nodes as possible, so as to minimize the number of computations in the network. Additional layers of hidden nodes may become necessary if the network is quite large and has a lot of training facts.

After building the network comes training the network. Code your favorite feed-forward training algorithm or use off-the-shelf programs like iBrainMaker (3) or iDynaMind (23), which both use the popular back propagation training algorithm.

Training a network is also more an art than a science. Adjusting learning parameters such as momentum, learning rate, and tolerance all become factors in how fast a network will converge (within some acceptable error range).

Testing the network follows training the network. Always save a percentage of the data for testing. Generally, 10–20 percent of the total number of vectors is sufficient to test the network. It is important to show the network data that it has not seen before, because only then will one know if the network is a good predictor, generalizer, or recognizer. Finally, good results will depend on the network definition and comparisons with other classifying techniques.

The last step in this process is running the network, presenting it with new input data, and gathering results. The first two steps above are the most difficult steps in the design process. As the saying goes, "A problem well-defined is a problem half solved."

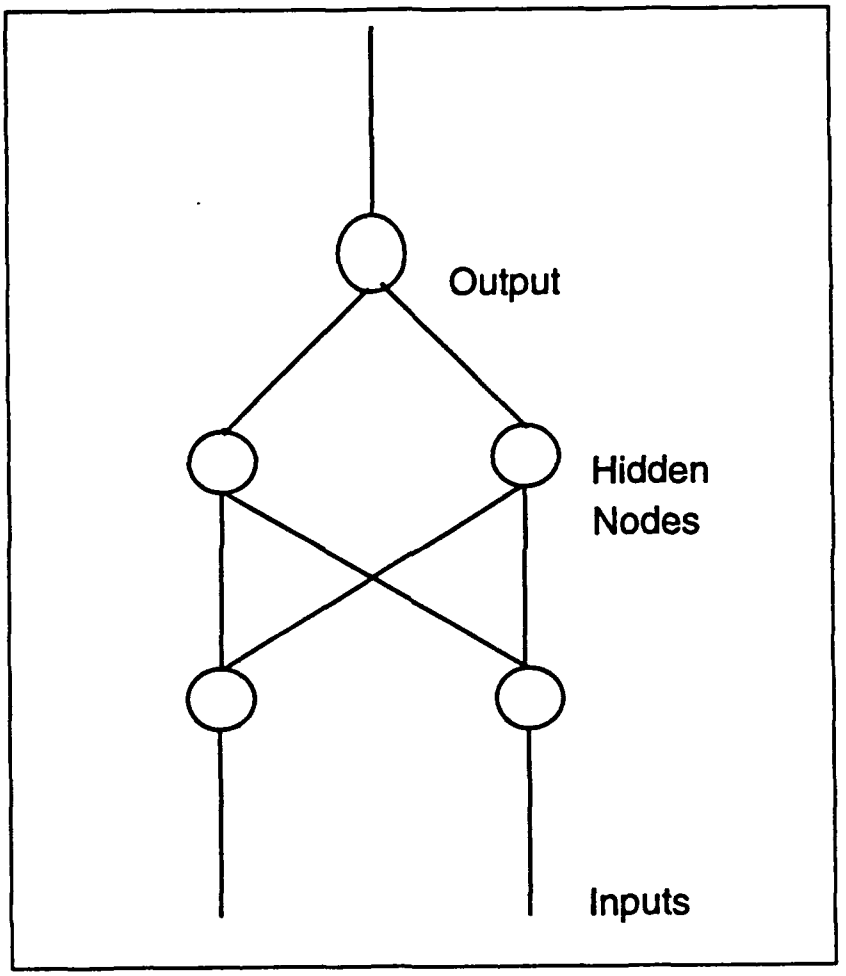

Figure 13. Simple Network

C.2 Summary

- Step 1. Define the problem. Decide whether you want the ANN to predict, generalize, or recognize.
- Step 2. Represent the Information. Decide what information will make good features for predictions, generalizations, or recognitions.
- Step 3. Get lots of data.
- Step 4. Define the network. Define the number of inputs, hidden nodes, outputs, and training data.
- Step 5. Train the network.
- Step 6. Test the network.
- Step 7. Run the Network.

Appendix D. *How to Convert GTRI Data to Simulator Formats and Shuffle*

D.1 Introduction

What follows below is the C code used in this research for data conversion and vector shuffling. Data conversion from GTRI format to iBrainMaker and Neural Graphics formats was necessary for thesis research. The shuffling programs are used to randomize the data vector order before training and testing a neural network.

D.2 How to Convert GTRI Data to iBrainMaker Format

```
/*============================================================
Purpose:  This program  reads  a GTRI data file and creates
an iBrainMaker fact file  for processing in iBrainMaker.
This particular program is limited to three classes,  but it
is easy to extend the ideas here for a file of several
classes.

Date: 5 June 1992

Author: Capt James Calvin
============================================================*/

#include<stdio.h>

/*=================
    Begin Main
=================*/

main()
{
  FILE *in_ptr, *out_ptr;
  int x, y, dimension;
  int vector[20];
  int class, junk1, junk2, junk3, junk4;
  int no_of_vectors;
  int temp_1;
  dimension = 17;
  no_of_vectors = 5795;
```

```
/*==================
    Open Files
====================*/

in_ptr = fopen("all30.d","r");
out_ptr = fopen("3-class_bm.dat","w");
fprintf(out_ptr,"Line# f1  f2  f3  f4  f4  f6  f7  f8  f9
f10  f11  f12  f13  f14  f15  f16  C1  C2  C3 \n");
  fscanf(in_ptr,"%d %d %d %d \n",&junk1,&junk2,&junk3,&junk4);
  for (x = 0; x < no_of_vectors; x++){
      for (y = 0; y < dimension; y++){
          fscanf(in_ptr,"%d",&temp_1);
          vector[y] = temp_1;
      }
      fscanf(in_ptr,"%d \n",&class);
      if (class == 1){
         for (y = 0; y < dimension; y++)
             fprintf(out_ptr," %d ",vector[y]);
         fprintf(out_ptr," 1 -1 -1 \n");
      }
     else if (class == 2){
          for (y = 0; y < dimension; y++)
              fprintf(out_ptr," %d ",vector[y]);
          fprintf(out_ptr," -1 1 -1 \n");
     }
     else if (class == 3){
          for (y = 0; y < dimension; y++)
              fprintf(out_ptr," %d ",vector[y]);
          fprintf(out_ptr," -1 -1 1 \n");
     }
  }
/*====================
     Close Files
======================*/

  fclose(in_ptr);
  fclose(out_ptr);

/*============================
     End of Main Program
==============================*/
}
```

D.3 How to Convert GTRI Data to Neural Graphics Format

```
/*================================================================
Purpose:  This program  reads GTRI data files and creates a Neural
Graphics fact file  for processing in Neural Graphics.  This
program reads in data from all30.d (all 30  classes of GTRI's data)
and then writes the data in Neural Graphics' format for three
classes.

Date: 5 June 1992

Author: Capt James Calvin
================================================================*/

#include<stdio.h>
#include<math.h>
#include<stdlib.h>

/*=================
    Begin Main
=================*/

main()
{
int x, y, dimension=17, no_of_vectors=5795;
int class, junk1, junk2, junk3, junk4;
FILE *in_ptr, *out_ptr;
int vector[20];
int temp_1;

/*============================
      Array Declaration
============================*/

int features[5795][20];

/*==================
     Open Files
==================*/

in_ptr = fopen("c-progs/all30.d' ,"r");
out_ptr = fopen("c-progs/ng_3class.dat","w+");
```

```
/*=======================
      Read Header
=======================*/

fscanf(in_ptr,"%d %d %d %d \n",&junk1,&junk2,&junk3,&junk4);

/*=======================
       Read Data
=======================*/

  for (x = 0; x < no_of_vectors; x++){
      for (y = 0; y < dimension; y++){
          fscanf(in_ptr,"%d",&temp_1);
          vector[y] = temp_1;
      }
      fscanf(in_ptr,"%d \n",&class);

/*================================
        Classify Data
================================*/

       if (class == 1)
          for (x = 0; x < 1; x++){
              for (y = 0; y < dimension; y++){
                  fprintf(out_ptr," %d ",vector[y]);
                  printf("I am after class 1 ID \n");
              }
          fprintf(out_ptr,"1 \n");
          }
       else if (class == 2)
            for (x = 0; x < 1; x++){
                for (y = 0; y < dimension; y++){
                    fprintf(out_ptr," %d ",vector[y]);
                    printf("I am after class 2 ID \n");
                }
                fprintf(out_ptr,"2 \n");
            }
       else if (class == 3)
            for (x = 0; x < 1;x++){
                for (y = 0; y < dimension; y++){
                    fprintf(out_ptr," %d ",vector[y]);
```

```
                    printf("I am after class 3 ID \n");
                }
                fprintf(out_ptr,"3 \n");
            }
        }

/*=======================
      Close Files
=======================*/

   fclose(in_ptr);
   fclose(out_ptr);

/*============================
     End of Main Program
============================*/
}
```

D.4 Shuffle Program for iBrainMaker Data Files

```
/*============================================================
Purpose: This program reads an iBrainMaker fact file and
shuffles the facts (vectors) so as to create a random fact
file for training in the BrainMaker simulator. This program
is limited to a three-class problem, but can be extended to
more classes with little modification.

Date: 6 June 1992

Author: Capt James Calvin
============================================================*/

#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define loop(A) for(i = 0; i < (A); i++)
float temp_1;
int iseed;
FILE *in_ptr, *out_ptr;

/*=================
    Begin Main
=================*/

main()
{

int x, y, i=0, j=0, idx=0, countall=0;
int vector[20], used[6000], order[6000];
int no_of_vectors = 5795;
int array_i;
int *array_prt[20];

/*============================
       Array Declaration
============================*/

int features[5795][20];
iseed = 2;
srand(iseed);
countall = 600;
```

```
/*==================
     Open Files
==================*/

/*--Open temp file (3-class_bm.dat) and open an output file
(3-class.bm.dat)--*/

if((in_ptr = fopen("c-progs/3-class_bm.dat","r"))==NULL)
   {
   printf("I'm not able to open the input file\n");
   exit(-1);
   }
if ((out_ptr=fopen("c-progs/3-class.bm.dat","w+"))==NULL)
   {
   printf("I'm not able to open the output file\n");
   exit(-1);
   }

rewind(in_ptr);

/*=======================================================
         Print a header for the output file
=======================================================*/

fprintf(out_ptr,"Line# f1  f2  f3  f4  f4  f6  f7  f8  f9
f10  f11  f12  f13  f14  f15  f16 C1  C2  C3 \n");

/*=======================================================
          Read all features into an array
=======================================================*/

for (x = 0; x < countall; x++)
   fscanf (in_ptr,"%d %d %d %d %d %d %d %d %d %d %d %d %d %d
    %d %d %d %d %d %d", &features [x][0],&features [x][1],
    &features[x][2],&features [x][3],&features [x][4],&features
    [x][5],&features [x][6],&features[x][7],&features [x][8],
    &features [x][9],&features [x][10],&features [x][11],
    &features[x][12], &features[x][13], &features[x][14],
    &features[x][15],&features[x][16],&features[x][17],
    &features[x][18],&features[x][19]);
```

```
/*=========================================================
       Randomize the order of the feature vectors
=========================================================*/

j = 0;
loop (countall){
     while{
          if(used[(idx = (int)((((float)rand())/RAND_MAX) *
          (countall - 1) + 0.5))]== 0)
                    {
                    used [idx] = 1;
                    printf("%d  \n",idx);
                    order [j] = idx;
                    j++;
                    break;
                    }
          }
}

/*=================================================
        Print random vectors to a file
=================================================*/

y = 0;
x = 0;
loop (countall)
  {
  x=order[y];
  fprintf (out_ptr, "%d %d %d %d %d %d %d %d %d %d %d %d %d
  %d %d %d  %d %d %d %d \n", features [x][0],features [x][1],
  features[x][2],features[x][3],features [x][4], features [x][5],
  features [x][6],features[x][7],features [x][8],features [x][9],
  features [x][10],features[x][11],features [x][12],features
  [x][13],features [x][14],features [x][15],features[x][16],
  features [x][17], features[x][18],features[x][19]);
  y++;
  }
```

```
/*============================
       Close Files
============================*/

fclose(in_ptr);
fclose(out_ptr);

/*============================
     End of Main Program
============================*/

}
```

D.5 Shuffle Program for Neural Graphics Data Files

```
/*============================================================
Purpose:  This program  reads  a Neural Graphic fact file and
shuffles the facts (vectors) so as to create a random fact file
for training in the Neural Graphic simulator.  This  program is
limited to a three-class problem, but can be extended to more
classes with little modification.

Date: 5 June 1992

Author: Capt James Calvin
============================================================*/

#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define loop(A) for(i = 0; i < (A); i++)
float temp_1;
int iseed;
FILE *in_ptr, *out_ptr;

/*====================
      Begin Main
====================*/

main()
{

int x, y, dimension=17, i=0, j=0, idx=0, count=0, countall=0;
int vector[20], used[6000], order[6000];
int no_of_vectors = 5795;
int array_i;
int *array_prt[20];

/*============================
      Array Declaration
============================*/

int features[5795][18];
iseed = 2;
srand(iseed);
countall =540;
```

```
count = 60;

/*====================
      Open Files
====================*/

/*--Open temp file (ng_3class.dat) and open an output file
(3-class_ng.dat)--*/

if((in_ptr = fopen("c-progs/ng_3class.dat","r")) == NULL){
   printf("I'm not able to open the input file\n");
   exit(-1);
   }
if((out_ptr = fopen("c-progs/3-class.ng.dat", "w+")) == NULL){
   printf("I'm not able to open the output file\n");
   exit(-1);
   }
rewind(in_ptr);

/*============================================================
            Print a header for the output file
============================================================*/

         fprintf(out_ptr,"%d %d 16 3 \n", countall, count);

/*============================================================
             Read all features into an array
============================================================*/

for (x = 0; x < countall; x++)
    fscanf (in_ptr,"%d %d %d %d %d %d %d %d %d %d %d %d %d %d
    %d %d %d %d", &features [x][0],&features [x][1],&features
    [x][2],&features[x][3], &features [x][4],&features [x][5],
    &features [x][6],&features[x][7],&features [x][8],
    &features [x][9],&features[x][10], &features[x][11],
    &features[x][12], &features[x][13],&features [x][14],
    &features[x][15], &features [x][16],&features[x][17]);
```

```
/*============================================================
          Randomize the order of the feature vectors
============================================================*/

j = 0;
loop (countall){
     while(1){
             if(used[(idx = (int)((((float)rand())/RAND_MAX) *
             (countall - 1) + 0.5))]== 0)
             {
             used [idx] = 1;
             printf("%d  \n",idx);
             order [j] = idx;
             j++;
             break;
             }
     }
}
/*============================================
          Print random vectors to a file
============================================*/

y = 0;
x = 0;
loop (countall){
     x=order[y];
     fprintf (out_ptr, "%d %d %d %d %d %d %d %d %d %d %d %d %d
     %d %d %d  %d %d \n", features [x][0],features [x][1],
     feature[x][2],features[x][3],features [x][4],features[x][5],
     features[x][6],features[x][7],features[x][8],features[x][9],
     features[x][10],features [x][11],features [x][12],features
     [x][13],features [x][14],features [x][15],features [x][16],
     features [x][17]);
     y++;
}
```

```
/*======================
        Close Files
======================*/

fclose(in_ptr);
fclose(out_ptr);

/*==========================
     End of Main Program
==========================*/
}
```

Appendix E. *C Programs to Normalize the Data*

E.1 Introduction

This appendix contains two C programs for normalizing the iBrainMaker and Neural Graphics feature vectors.

E.2 Normalization and Dynamic Ranging for iBrainMaker Format

```
/*=============================================================*/
/*=====
/*=====   THIS PROGRAM WAS WRITTEN BY DON WILLIS, GE091D, AND IT
/*=====   WAS MODIFIED BY JAMES CALVIN, GE92D.  THE PROGRAM
/*=====   STATISTICALLY NORMALIZES THE DATA SETS ACCORDING TO A
/*=====   MEAN AND VARIANCE GAUSSIAN DISTRTIBUTION
/*=====
/*=============================================================*/
/*=============================================================*/
/*===  System includes  =======================================*/
/*=============================================================*/

#include <stdio.h>
#include <math.h>

/*=============================================================*/
/*===  Defines  ===============================================*/
/*=============================================================*/

#define INPUTS 75 /* max number of features */
/*=============================================================*/
/*===  Main Routine  ==========================================*/
/*=============================================================*/

void main (argc,argv)
int argc;
char *argv[];
{
/*=================================================*/
/*=== local variables                          ===*/
/*=================================================*/
```

```
FILE *input;
char infile[50];

FILE *output;
char outfile[50];
float trash, value, max = 0;
char *s;
float trash,value;
double max1=0,max2=0,max3=0,max4=0,max5=0;
double max6=0,max7=0,max8=0,max9=0,max10=0,max11=0;
double max12=0,max13=0,max14=0,max15=0,max16=0;
double f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14;
double f15,f16;
float deviation[INPUTS], average[INPUTS];
int i, j, inputs, outputs, ivalue;
int count1, count2, count, waste, temp;
float value1, value2, value3;

printf("Enter the number of inputs then the number of
  outputs.\n");

scanf("%d %d", &inputs, &outputs);

printf("Enter total number of training/testing vectors.
  \n");

scanf("%d", &count);

fflush(stdin);

/*=====================================================*/
/*=== did user specify an input file              ====*/
/*=====================================================*/

if (argc != 2)
{
  printf ("\n\nUsage -> stat-norm <filename>\n\n");
/*=== exit after pointing out the error ===*/
  exit (1000);
}
```

```
/*=========================================================*/
/*=== user did specify an input file                    ===*/
/*=========================================================*/

strcpy (infile, argv[1]);/* use inputted name as base */

/*=========================================================*/
/*=== Open Input File                                   ===*/
/*=========================================================*/

printf ("\nOpening Input File:  %s\n\n", infile);
if (!(input = fopen(infile, "rb"))){
   printf ("\nCan't open input file: %s\n\n", infile);
   exit (2000);
}
if (inputs < 0 || outputs < 0){
   printf ("One of the header inputs is negative\n\n");
   exit (3000);
}
printf ("There are %d training/testing vectors\n", count);
printf ("There are %d inputs\n", inputs);
printf ("There are %d outputs\n\n", outputs);

/*=========================================================*/
/*=== initialize things                                 ===*/
/*=========================================================*/

for (i = 0; i < inputs; i++)
    average[i] = deviation[i] = 0.0;

/*=========================================================*/
/*=== loop until all data has been read in              ===*/
/*=========================================================*/

printf ("Reading the Data\n\n");
fscanf (input,"%s %s %s %s %s %s %s %s %s %s %s %s %s %s %s
 %s %s %s %s %s \n", &s, &s, &s, &s, &s, &s, &s, &s, &s, &s,
 &s, &s, &s, &s, &s, &s, &s, &s, &s, &s);

fflush(stdin);
for (i = 0; i < count; i++){
```

```
    fscanf (input, "%d ", &trash);/* read line counter */
            for (j = 0; j < inputs; j++){
        fscanf (input, "%f ", &value);
        average[j] += value;
    }
    fscanf (input, "%d %d %d \n", &trash, &trash, &trash);
}
fclose (input);

/*=====================================================*/
/*=== calculate the averages                        ===*/
/*=====================================================*/

printf ("Calculating Averages\n\n");
for (i = 0; i < inputs; i++)
    average[i] /= (float)count;

/*=====================================================*/
/*=== Re-open the input file                        ===*/
/*=====================================================*/

printf ("Re-Opening Input File:  %s\n\n", infile);
if (!(input = fopen(infile, "rb"))){
   printf ("\nCan't re-open input file: %s\n\n", infile);
   exit (2000);
}

/*=====================================================*/
/*=== loop until all data has been read in          ===*/
/*=====================================================*/

fscanf(input,"%s %s %s %s %s %s %s %s %s %s %s %s %s %s %s
 %s %s %s %s %s \n", &s, &s, &s, &s, &s, &s, &s, &s, &s,
 &s,&s,&s,&s,&s,&s,&s,&s,&s,&s);

fflush(stdin);
printf ("Reading the Data\n\n");
for (i = 0; i < count; i++){
    fscanf (input, "%d ", &trash);
    for (j = 0; j < inputs; j++){
        fscanf (input, "%f ", &value);
        value -= average[j];
```

```
        value *= value;
                deviation[j] += value;
    }
    fscanf (input, "%d %d %d \n", &trash, &trash, &trash);
}
fclose (input);

/*=======================================================*/
/*=== calculate the standard deviation              ===*/
/*=======================================================*/

printf ("Calculating Standard Deviations\n\n");
for (i = 0; i < inputs; i++){
    deviation[i] /= count - 1;
    deviation[i]  = (float)sqrt((double)deviation[i]);
}

/*=======================================================*/
/*===      make output-file name                    ===*/
/*=======================================================*/

sprintf (outfile, "%s.sn", argv[1]);

/*=======================================================*/
/*=== Open Output File                              ===*/
/*=======================================================*/

printf ("Opening Output File:  %s\n\n", outfile);
if (!(output = fopen(outfile, "wb"))){
    printf ("\nCan't open output file: %s\n\n", outfile);
    exit (2000);
}

/*=======================================================*/
/*=== Re-open the input file                        ===*/
/*=======================================================*/

printf("Re-Opening Input File (last time):%s\n\n",infile);
if (!(input = fopen(infile, "rb"))){
    printf ("\nCan't re-open input file: %s\n\n", infile);
    exit (2000);
}
```

```
/*=====================================================*/
/*=== read data in, modify it, save it back out    ===*/
/*=====================================================*/

printf ("Reading, Modifying and Re-Saving the Data\n\n");
fscanf (input,"%s %s %s %s %s %s %s %s %s %s %s %s %s %s %s
 %s %s %s %s %s \n", &s, &s, &s, &s, &s, &s, &s, &s, &s, &s,
 &s, &s, &s, &s, &s, &s, &s, &s, &s, &s);

fflush(stdin);
for (i = 0; i < count; i++){
    fscanf  (input,"%d",&ivalue);
    fprintf (output,"%d",ivalue);
            for (j = 0; j < inputs; j++){
        fscanf(input,"%f",&value);
        value -= average[j];
        value /= deviation[j];
        fprintf(output,"%f",value);
    }
    fscanf(input,"%d %d %d \n",&ivalue1,&ivalue2,&ivalue3);
    fprintf(output,"%d %d %d \n",ivalue1,ivalue2,ivalue3);
}
fflush(output);

/*=====================================================*/
/*===Normalize Data Between -1 and 1==================*/
/*=====================================================*/

/*=====================================================*/
/*=== make output-file name                       ===*/
/*=====================================================*/

sprintf (infile, "%s.sn", argv[1]);
sprintf (outfile, "%s.n", argv[1]);

/*=====================================================*/
/*=== Open Output File                            ===*/
/*=====================================================*/

printf ("Opening Output File:  %s\n\n", outfile);
if (!(output = fopen(outfile, "wb"))){
```

```
   printf ("\nCan't open output file: %s\n\n", outfile);
   exit (2000);
}

/*=====================================================*/
/*=== Re-open the input file                        ===*/
/*=====================================================*/

printf ("Re-Opening Input File (last time):%s\n\n",infile);
if (!(input = fopen(infile, "rb"))){
   printf ("\nCan't re-open input file: %s\n\n", infile);
   exit (2000);
}

/*=====================================================*/
/*=== read data in, modify it, save it back out     ===*/
/*=====================================================*/


printf ("Reading the Data\n\n");
for (i = 0; i < count; i++){
    fscanf (input, "%d ", &ivalue);
    fscanf (input, "%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf
    %lf %lf %lf %lf %lf %lf", &f1,&f2,&f3,&f4,&f5,&f6,
    &f7,&f8,&f9,&f10,&f11,&f12,&f13,&f14,&f15,&f16);
    if (fabs(f1) > max1)
       max1 = fabs(f1);
    if (fabs(f2) > max2)
       max2 = fabs(f2);
    if ( fabs(f3) > max3)
       max3 =  fabs(f3);
    if (fabs(f4) > max4)
       max4 =  fabs(f4);
    if (fabs(f5) > max5)
       max5 =  fabs(f5);
    if (fabs(f6) > max6)
       max6 = fabs(f6);
    if (fabs(f7) > max7)
       max7 = fabs(f7);
    if (fabs(f8) > max8)
       max8 = fabs(f8);
    if (fabs(f9) > max9)
```

```
        max9 = fabs(f9);
    if (fabs(f10) > max10)
        max10 = fabs(f10);
    if (fabs(f11) > max11)
        max11 = fabs(f11);
    if (fabs(f12) > max12)
        max12 = fabs(f12);
    if (fabs(f13) > max13)
        max13 = fabs(f13);
    if (fabs(f14) > max14)
        max14 = fabs(f14);
    if (fabs(f15) > max15)
        max15 = fabs(f15);
    if (fabs(f16) > max16)
        max16 = fabs(f16);
    fscanf  (input,"%f %f %f \n", &value1,&value2,
     &value3);
}
fseek(input,0,0);

/*=====================================================*/
/*=== read data in, modify it, save it back out    ===*/
/*=====================================================*/

printf ("Reading, Modifying and Re-Saving the Data\n\n");

/*****************************************************************
*         Printing a header for a 3-class problem
*****************************************************************/

fprintf(output,"Line# f1  f2  f3  f4  f5  f6  f7  f8  f9
        f10  f11  f12  f13  f14  f15  f16  C1  C2  C3 \n");
for (i = 0; i < count; i++){
    fscanf  (input,  "%d ", &ivalue);
    fprintf (output, "%d ",  ivalue);
    fscanf (input, "%lf %lf %lf %lf %lf %lf %lf %lf %lf
    %lf %lf %lf %lf %lf %lf %lf", &f1,&f2,&f3,&f4,&f5,
    &f6, &f7, &f8, &f9, &f10, &f11, &f12, &f13, &f14,
    &f15, &f16);
    f1 /= max1;
    if (f1 >= 0.0)
       f1 += 0.0005;
```

```
else
   f1 -+ 0.0005;
f2 /= max2;
if (f2 >= 0.0)
   f2 += 0.0005;
else
   f2 -+ 0.0005;
f3 /= max3;
if (f3 >= 0.0)
   f3 += 0.0005;
else
   f3 -+ 0.0005;
f4 /= max4;
if (f4 >= 0.0)
   f4 += 0.0005;
else
   f4 -+ 0.0005;
f5 /= max5;
if (f5 >= 0.0)
   f5 += 0.0005;
else
   f5 -+ 0.0005;
f6 /= max6;
if (f6 >= 0.0)
   f6 += 0.0005;
else
   f6 -+ 0.0005;
f7 /= max7;
if (f7 >= 0.0)
   f7 += 0.0005;
else
   f7 -+ 0.0005;
f8 /= max8;
if (f8 >= 0.0)
   f8 += 0.0005;
else
   f8 -+ 0.0005;
f9 /= max9;
if (f9 >= 0.0)
   f9 += 0.0005;
else
   f9 -+ 0.0005;
```

```
f10 /= max10;
if (f10 >= 0.0)
   f10 += 0.0005;
else
   f10 -+ 0.0005;
f11 /= max11;
if (f11 >= 0.0)
   f11 += 0.0005;
else
   f11 -+ 0.0005;
f12 /= max12;
if (f12 >= 0.0)
   f12 += 0.0005;
else
   f12 -+ 0.0005;
f13 /= max13;
if (f13 >= 0.0)
   f13 += 0.0005;
else
   f13 -+ 0.0005;
f14 /= max14;
if (f14 >= 0.0)
   f14 += 0.0005;
else
   f14 -+ 0.0005;
f15 /= max15;
if (f15 >= 0.0)
   f15 += 0.0005;
else
   f15 -+ 0.0005;
f16 /= max16;
if (f16 >= 0.0)
   f16 += 0.0005;
else
   f16 -+ 0.0005;

fprintf (output, " %.3lf %.3lf %.3lf %.3lf %.3lf
%.3lf %.3lf %.3lf %.3lf %.3lf %.3lf %.3lf %.3lf
%.3lf %.3lf %.3lf ", f1,f2,f3,f4,f5 ,f6,f7,f8 ,f9,
f10,f11,f12,f13,f14,f15,f16);
fscanf  (input,  "%f %f %f \n", &value1, &value2,
&value3);
```

```
   fprintf (output, "%.1f %.1f %.1f \n", value1,value2,
   value3);
}

fclose (input);
fclose (output);

/*=======================================================*/
/*=== we're done                                      ===*/
/*=======================================================*/

printf ("Finished.\n\n");
}

/*=============================================================*/
/*=============================================================*/
/*=============================================================*/
```

E.3 Normalization and Dynamic Ranging for Neural Graphics Format

```
/*=============================================================
/*=====
/*===== THIS PROGRAM WAS ORIGINALLY WRITTEN BY DON WILLIS, GE091D,
/*===== AND IT  WAS MODIFIED BY JAMES CALVIN, GE92D.  THE PROGRAM
/*=====   STATISTICALLY NORMALIZES THE DATA SETS ACCORDING TO A
/*=====   MEAN AND VARIANCE GAUSSIAN DISTRIBUTION, AND THEN IT RE-
/*===== DUCES THE DYNAMIC RANGE TO -1.0 TO 1.0.
/*=====
/*=============================================================
/*=============================================================
/*===  System includes  =======================================
/*=============================================================*/

#include <stdio.h>
#include <math.h>

/*============================*/
/*===  Defines  ==============*/
/*============================*/

#define INPUTS 75; /* max number of features */

/*============================*/
/*===  Main Routine  =========*/
/*============================*/

void main (argc,argv)
int argc;
char *argv[];
{
/*=====================================================*/
/*=== local variables                              ===*/
/*=====================================================*/

FILE *fopen();
FILE *input, *fopen();
char infile[50];
FILE *output;
char outfile[50];
float trash,value;
```

```
double max1=0,max2=0,max3=0,max4=0,max5=0;
double max6=0,max7=0,max8=0,max9=0,max10=0,max11=0;
double max12=0,max13=0,max14=0,max15=0,max16=0;
double f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14;
double f15,f16;
float deviation[INPUTS], average[INPUTS];
int i, j, inputs, outputs, ivalue;
int count1, count2, count, waste, temp;

/*=======================================================*/
/*=== did user specify an input file              ====*/
/*=======================================================*/

if (argc != 2)
{
   printf ("\n\nUsage -> stat-norm <filename>\n\n");
   /*=== exit after pointing out the error ===*/
   exit (1000);
}

/*=======================================================*/
/*=== user did specify an input file              ===*/
/*=======================================================*/

strcpy (infile, argv[1]);/* use inputted name as base */

/*=======================================================*/
/*=== Open Input File                             ===*/
/*=======================================================*/

printf ("\nOpening Input File:  %s\n\n", infile);
if (!(input = fopen(infile, "rb")))
{
   printf ("\nCan't open input file: %s\n\n", infile);
   exit (2000);
}

/*=======================================================*/
/*=== read the header information                 ===*/
/*=======================================================*/

fscanf(input,"%d %d %d %d\n",&count1,&count2,&inputs,
```

```
  &outputs);
if(count1 < 0 || count2 < 0 || inputs < 0 || outputs < 0)
{
   printf ("One of the header inputs is negative\n\n");
   exit (3000);
}
printf ("There are %d training vectors\n", count1);
printf ("There are %d test vectors\n", count2);
printf ("There are %d inputs\n", inputs);
printf ("There are %d outputs\n\n", outputs);
count = count1 + count2;

/*=========================================================*/
/*=== initialize things                                 ===*/
/*=========================================================*/

for (i = 0; i < inputs; i++)
{
   average[i] = deviation[i] = 0.0;
}

/*=========================================================*/
/*=== loop until all data has been read in              ===*/
/*=========================================================*/

printf ("Reading the Data\n\n");
for (i = 0; i < count; i++)
{
   fscanf(input,"%d ",&trash);/* read line counter */
   for (j = 0; j < inputs; j++)
   {
     fscanf(input,"%f ",&value);/* read float values */
     average[j] += value;
   }
   fscanf (input, "%d \n", &ivalue);
}
fclose (input);

/*=========================================================*/
/*=== calculate the averages                            ===*/
/*=========================================================*/
```

```
printf ("Calculating Averages\n\n");
for (i = 0; i < inputs; i++)
{
   average[i] /= (float)count;
}

/*=====================================================*/
/*=== Re-open the input file                        ===*/
/*=====================================================*/

printf ("Re-Opening Input File:  %s\n\n", infile);
if (!(input = fopen(infile, "rb")))
{
   printf ("\nCan't re-open input file: %s\n\n", infile);
   exit (2000);
}

/*=====================================================*/
/*=== throw away the header information this time   ===*/
/*=====================================================*/

fscanf(input,"%d %d %d %d\n",&waste,&waste,&waste,&waste);

/*=====================================================*/
/*=== loop until all data has been read in          ===*/
/*=====================================================*/

printf ("Reading the Data\n\n");
for (i = 0; i < count; i++)
{
   fscanf (input, "%d ", &trash);/* read line counter */
   for (j = 0; j < inputs; j++)
   {
     fscanf (input,"%f",&value);/* read values */
     value -= average[j]; /*subtract off the average*/
     value *= value; /* square the result */
     deviation[j] += value;/*hang onto it until all done*/
   }
   fscanf (input, "%d \n", &ivalue);
}
fclose (input);
```

```
/*=====================================================*/
/*=== calculate the standard deviation              ===*/
/*=====================================================*/

printf ("Calculating Standard Deviations\n\n");

for (i = 0; i < inputs; i++)
{
   deviation[i] /= count - 1;
   deviation[i]  = (float)sqrt((double)deviation[i]);
}

/*=====================================================*/
/*=== make output-file name                         ===*/
/*=====================================================*/

sprintf (outfile, "%s.sn", argv[1]);

/*=====================================================*/
/*=== Open Output File                              ===*/
/*=====================================================*/

printf ("Opening Output File:  %s\n\n", outfile);
if (!(output = fopen(outfile, "wb")))
{
   printf ("\nCan't open output file: %s\n\n", outfile);
   exit (2000);
}

/*=====================================================*/
/*=== Re-open the input file                        ===*/
/*=====================================================*/

printf("Re-Opening Input File (last time):%s\n\n",infile);
if (!(input = fopen(infile, "rb")))
{
   printf ("\nCan't re-open input file: %s\n\n", infile);
   exit (2000);
}

/*=====================================================*/
/*=== read and save header                          ===*/
```

```
/*=======================================================*/

fscanf(input,"%d %d %d %d\n",&count1,&count2,&inputs,
  &outputs);
fprintf(output,"%d %d %d %d\n",count1,count2,inputs,
  outputs);

/*=======================================================*/
/*=== read data in, modify it, save it back out     ===*/
/*=======================================================*/

printf("Reading, Modifying and Re-Saving the Data\n\n");
for (i = 0; i < count; i++)
{
   fscanf(input,"%d ",&ivalue);/*read line counter*/
   fprintf(output,"%d ",ivalue);/*save line counter*/
   for (j = 0; j < inputs; j++)
   {
     fscanf(input,"%f",&value);/* read float value */
     value -= average[j];/* modify the value */
     value /= deviation[j];
     fprintf(output,"%f",value);/*save modified value*/
   }
   fscanf  (input,  "%d \n", &ivalue);
           fprintf (output, "%d \n",  ivalue);
}
fflush(output);

/*=======================================================*/
/*===Normalize Data Between -1 and 1==================*/
/*=======================================================*/

/*=======================================================*/
/*=== make output-file name                         ===*/
/*=======================================================*/

sprintf (infile, "%s.sn", argv[1]);
sprintf (outfile, "%s.n", argv[1]);

/*=======================================================*/
/*=== Open Output File                              ===*/
/*=======================================================*/
```

```
printf ("Opening Output File:  %s\n\n", outfile);
if (!(output = fopen(outfile, "wb")))
{
   printf ("\nCan't open output file: %s\n\n", outfile);
   exit (2000);
}

/*=====================================================*/
/*=== Re-open the input file                        ===*/
/*=====================================================*/

printf("Re-Opening Input File (last time):%s\n\n",infile);
if (!(input = fopen(infile, "rb")))
{
   printf ("\nCan't re-open input file: %s\n\n", infile);
   exit (2000);
}

/*=====================================================*/
/*=== read the header                               ===*/
/*=====================================================*/

fscanf(input,"%d %d %d %d\n",&count1,&count2,&inputs,
  &outputs);

/*=====================================================*/
/*=== read data in, modify it, save it back out     ===*/
/*=====================================================*/
printf ("Reading, Modifying and Re-Saving the Data\n\n");
for (i = 0; i < count; i++){
    fscanf (input, "%d ", &ivalue); /* read line counter */
    fscanf (input, "%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf
     %lf %lf %lf %lf %lf %lf", &f1,&f2,&f3,&f4,&f5,&f6,
     &f7,&f8,&f9,&f10,&f11,&f12,&f13,&f14,&f15,&f16);
    if (fabs(f1) > max1)
       max1 = fabs(f1);
    if (fabs(f2) > max2)
       max2 = fabs(f2);
    if (fabs(f3) > max3)
       max3 =  fabs(f3);
    if (fabs(f4) > max4)
```

```
        max4 =  fabs(f4);
      if (fabs(f5) > max5)
         max5 =  fabs(f5);
      if (fabs(f6) > max6)
         max6 = fabs(f6);
      if (fabs(f7) > max7)
         max7 = fabs(f7);
      if (fabs(f8) > max8)
         max8 = fabs(f8);
      if (fabs(f9) > max9)
         max9 = fabs(f9);
      if (fabs(f10) > max10)
         max10 = fabs(f10);
      if (fabs(f11) > max11)
         max11 = fabs(f11);
      if (fabs(f12) > max12)
         max12 = fabs(f12);
      if (fabs(f13) > max13)
         max13 = fabs(f13);
      if (fabs(f14) > max14)
         max14 = fabs(f14);
      if (fabs(f15) > max15)
         max15 = fabs(f15);
      if (fabs(f16) > max16)
         max16 = fabs(f16);
      fscanf (input,"%d \n", &ivalue);
          }
          fseek(input,0,0);

/*=========================================================*/
/*=== read and save header                              ===*/
/*=========================================================*/

fscanf(input,"%d %d %d %d\n",&count1,&count2,&inputs,
  &outputs);
fprintf(output,"%d %d %d %d\n",count1,count2,inputs,
 outputs);

/*=========================================================*/
/*=== read data in, modify it, save it back out        ===*/
/*=========================================================*/
```

```
printf ("Reading, Modifying and Re-Saving the Data\n\n");

for (i = 0; i < count; i++){
    fscanf  (input,  "%d ", &ivalue);/*read line counter*/
    fprintf (output, "%d ",  ivalue);/*save line counter*/
    fscanf (input, "%lf %lf %lf %lf %lf %lf %lf %lf %lf
    %lf %lf %lf %lf %lf %lf %lf", &f1,&f2,&f3,&f4,&f5,&f6,
    &f7, &f8, &f9, &f10, &f11, &f12, &f13, &f14, &f15, &f16);
    f1 /= max1;
    if (f1 >= 0.0)
       f1 += 0.0005;
    else
       f1 -+ 0.0005;
    f2 /= max2;
    if (f2 >= 0.0)
       f2 += 0.0005;
    else
       f2 -+ 0.0005;
    f3 /= max1;
    if (f3 >= 0.0)
       f3 += 0.0005;
    else
       f3 -+ 0.0005;
    f4 /= max1;
    if (f4 >= 0.0)
       f4 += 0.0005;
    else
       f4 -+ 0.0005;
    f5 /= max1;
    if (f5 >= 0.0)
       f5 += 0.0005;
    else
       f5 -+ 0.0005;
    f6 /= max1;
    if (f6 >= 0.0)
       f6 += 0.0005;
    else
       f6 -+ 0.0005;
    f7 /= max1;
    if (f7 >= 0.0)
       f7 += 0.0005;
```

```
else
   f7 -+ 0.0005;
f8 /= max1;
if (f8 >= 0.0)
   f8 += 0.0005;
else
   f8 -+ 0.0005;
f9 /= max1;
if (f9 >= 0.0)
   f9 += 0.0005;
else
   f9 -+ 0.0005;
f10 /= max1;
if (f10 >= 0.0)
   f10 += 0.0005;
else
   f10 -+ 0.0005;
f11 /= max1;
if (f11 >= 0.0)
   f11 += 0.0005;
else
   f11 -+ 0.0005;
f12 /= max1;
if (f12 >= 0.0)
   f12 += 0.0005;
else
   f12 -+ 0.0005;
f13 /= max1;
if (f13 >= 0.0)
   f13 += 0.0005;
else
   f13 -+ 0.0005;
f14 /= max1;
if (f14 >= 0.0)
   f14 += 0.0005;
else
   f14 -+ 0.0005;
f15 /= max1;
if (f15 >= 0.0)
   f15 += 0.0005;
else
   f15 -+ 0.0005;
```

```
    f16 /= max1;
    if (f16 >= 0.0)
       f16 += 0.0005;
    else
       f16 -+ 0.0005;

    fprintf (output, " %.3lf %.3lf %.3lf %.3lf %.3lf
    %.3lf %.3lf %.3lf %.3lf %.3lf %.3lf %.3lf %.3lf
    %.3lf %.3lf %.3lf ",f1,f2,f3,f4,f5,f6, f7,f8,f9,
    f10,f11,f12,f13,f14,f15,f16);
    fscanf  (input,  " %d \n", &ivalue);
    fprintf (output, " %d \n",  ivalue);
}
fclose (input);
fclose (output);
/*=========================================================*/
/*=== we're done                                       ===*/
/*=========================================================*/

printf ("Finished.\n\n");
}
/*=============================================================*/
/*=============================================================*/
/*=============================================================*/
```

Appendix F. *Neural Graphics Module Modification for ETANN Chip Simulation*

F.1 Introduction

This appendix contains the backpropagation C module, jcbackprop.c, for Neural Graphics that was modified to emulate the sigmoidal transfer function and to limit weight values to within ETANN's dynamic range. The Neural Graphics executable file name was changed from *net* to *jcnet*. Hence, *jcbackprop.c* along with modified files *saver.c*, *general.c*, and *makeinput.c* are all the files affected by the ETANN emulation. *General.c* and *saver.c* modifications were presented in Appendix B. Below are the modifications to *makeinput.c* and *backprop.c* (note: *backprop.c* was replaced with *jcbackprop.c* which is included below).

```
Replace "case CLASS" in makeinput.c with the following changes:
/*********************************************************************
case CLASS    : loopi(output){ if(i==ctype)
                                  doft[parameters.output-1-i] = 0.89;
               else
         doft[parameters.output-1-i] = -0.94;}

*********************************************************************/
Replace backprop.c with the following module and either rename the
file as jcbackprop.c (requires changing the makefile: renaming
executable command and adding jcbackprop.c) or leave alone and note
that the simulator is now operating only as an ETANN simulator.
```

```
/*****************************************************************
*
*       DATE: 1 Aug 1992
*       VERSION: 3
*
*       NAME: Module for Backprop
*       MODULE NUMBER: 1.6
*       DESCRIPTION: Using output, adjusts weights, & reduce error.
*       ALGORITHM: Modified Werbos Multilayer Perceptron Back
*                  propagation.
*       PASSED VARIABLES: None
*       RETURNS: None
*       GLOBAL VARIABLES USED: Weight Vectors
*       GLOBAL VARIABLES CHANGED: Weight Vectors
*       FILES READ: None
*       FILES WRITTEN: None
*       HARDWARE INPUT: None
*       HARDWARE OUTPUT: None
*       MODULES CALLED: None
*       CALLING MODULES: Main Loop
*
*       AUTHOR: Gregory L. Tarr
*       HISTORY: Modified by Capt James Calvin for ETANN chip
*                Simulation
*
*****************************************************************/

#include "definitions.h"
#include <math.h>

extern nnet *net;
extern setup parameters;
extern examplars, count;
extern void TRAIN_NULL();
float calcy(),calcm(),fixy(),calcy_linear();
float calcy_nomask(),calcy_linear_nomask();
float hardness = 1.0;
void TRAIN_LOWER_LAYER(),TRAIN_OUTPUT_LAYER();
init_train_BACKPROP(){
  int i;
  init_net(parameters);
  loopi(parameters.layers){
```

```
    net->layer1[i]->updater = TRAIN_LOWER_LAYER;
    net->layer1[i]->eta[0] = 0.3;
    net->layer1[i]->eta[1] = 0.7;
  }
  net->layer1[parameters.layers-1]->updater=TRAIN_OUTPUT_LAYER;
  loopi(parameters.layers)
  net->layer1[i]->propagate = calcy;
}
void TRAIN_OUTPUT_LAYER (l_on,l_up)
layer *l_on,*l_up;{
  int i,j,k,lower,upper;
  float neta,momentum;
  lower = l_on->size_input;
  upper = l_on->size_output;
  neta     = l_on->eta[0]/(float)l_on->size_input;
  momentum = l_on->eta[1];
  dely(l_on->output,net->doft,l_on->delta,upper);

/**Bias weights constrained between -15 and 15 for 6 ETANN biases**/

  loopj(upper) {
      if (fabs((double)l_on->theta[j]) <= 15.0)
l_on->theta[j] =  l_on->theta[j] + neta * l_on->delta[j] ;
      if (fabs((double)l_on->theta[j]) > 15.0){
if (l_on->theta[j] >= 0.0) l_on->theta[j] = 15.0;
if (l_on->theta[j] < 0.0) l_on->theta[j] = -15.0;
      }
    loopi(l_on->size_input){

/***Weights constrained between -2.5 and 2.5 for ETANN chip***/

    if (fabs((double)l_on->weights[j][i]) <= 2.50)
       l_on->weights[j][i] +=
            neta * l_on->delta[j] * l_on->input[i] * l_on->mask[i];
    if (fabs((double)l_on->weights[j][i]) > 2.50){
       if (l_on->weights[j][i] >= 0.0) l_on->weights[j][i] = 2.50;
       if (l_on->weights[j][i] < 0.0) l_on->weights[j][i] = -2.50;
      }
    }
  }
}
void TRAIN_LOWER_LAYER(l_on,l_up)
```

```
layer *l_on,*l_up;{
  int i,j,k,lower,upper;
  float neta,momentum;
  lower = l_on->size_input;
  upper = l_on->size_output;
  neta = l_on->eta[0]/l_on->size_input;
  momentum = l_on->eta[1];
  delx(l_on->output,l_up->delta,l_on->delta,
    l_up->weights,l_up->size_output,upper);

/**Bias weights constrained between -15 and 15 for 6 ETANN biases**/

  loopj(upper) {
      if (fabs((double)l_on->theta[j]) <= 15.0)
l_on->theta[j] =  l_on->theta[j] + neta * l_on->delta[j] ;
      if (fabs((double)l_on->theta[j]) > 15.0){
if (l_on->theta[j] >= 0.0) l_on->theta[j] = 15.0;
if (l_on->theta[j] < 0.0) l_on->theta[j] = -15.0;
     }
    loopi(l_on->size_input) {

/*****Weights constrained between -2.5 and 2.5 for ETANN chip *****/

      if (fabs((double)l_on->weights[j][i]) <= 2.50)
         l_on->weights[j][i] += neta * l_on->delta[j] *
 l_on->input[i] * l_on->mask[i];
      if (fabs((double)l_on->weights[j][i]) > 2.50){
 if (l_on->weights[j][i] >= 0.0) l_on->weights[j][i] = 2.50;
 if (l_on->weights[j][i] < 0.0) l_on->weights[j][i] = -2.50;
      }
    }
  }
}
 float calcy(j,l_on)
 int j;
 layer *l_on;{
   int i,k,number;
   float y;
   return fixy(calcy_linear(j,l_on),1.0);}
   float calcy_linear(j,l_on)
   int j;
   layer *l_on;{
```

```
    int i;
    float y;
    y = 0.0;
    loopi(l_on->size_input){ if(l_on->mask[i] == 0.0) continue;
          y +=l_on->input[i] * l_on->weights[j][i];
  }
  y += l_on->theta[j];
  return y;
}

float calcy_nomask(j,l_on)
int j;
layer *l_on;{
  int i,k,number;
  float y;
  y = calcy_linear_nomask(j,l_on);
  return fixy(calcy_linear(j,l_on),1.0);
}
float calcy_linear_nomask(j,l_on)
int j;
layer *l_on;{
  int i,k,number;
  float y;
  y = 0.0;
  loopi(l_on->size_input) {
    y = y + l_on->input[i] * l_on->weights[j][i];
  }
  y = y + l_on->theta[j];
  return y;
}
/*****************************************************************
*        Modified Algorithm for ETANN Chip Simulation
*****************************************************************/
float fixy(y,hardness)
float y, hardness ;
{
  float xx;
  int i;
  hardness = 1.74;
  if (y >  8.0) return .89;
  if (y < -8.0) return -.94;
  return( 1.83/(1.0 + (float)exp(-(double)(hardness * y))) - .94);
```

```
}
/*****************************************************************/
dely(y,doft,del,output) /* computer error vector for output  */
float *y,*doft,*del; /* Interface requires computed output   */
int output;    /* Desired output and storage for error: del. */
{
  int i;
  if(parameters.output_data == 1)
    loopi(output) del[i] = 0.1*(doft[i]-y[i]);
  else
    /**** Derivative of ETANN transfer function ****/
  loopi(output) del[i] = 0.95*(.94 + y[i])*(.89 - y[i])*(doft[i]-y[i]);
}
dely_linear(y,doft,del,output)/*computer error vector for output*/
float *y,*doft,*del;/* Interface requires computed output */
int output; /* Desired output and storage for error: del. */
{
  int i;
  loopi(output) del[i] = (doft[i]-y[i])/(float)output;
}
delx(x,del,del_down,w,upper,lower)
float *x,*del,*del_down,**w;
int upper,lower;
{
  float sum;
  int i,j;
  loopj(lower){
    sum = 0.0;
    loopi(upper){
      sum = sum + del[i] * w[i][j];
    }
    /**** Derivative of ETANN transfer function ****/
    sum = 0.95*(.94 + x[j])*(.89 - x[j]) * sum;
    del_down[j] = sum;
  }
}
check_linear(){
  if(parameters.output_data == 1)
  {
    net->layer1[parameters.layers-1]->propagate = calcy_linear;
  }
}
```

Appendix G. *ETANN Characterization Tools*

G.1 Introduction

The C programs in this appendix are only a sampling of the several programs written to perform sigmoidal characterization of ETANN neurons. The general test procedures were as follows: 1) run test programs to get data from eight ETANN chips while varying the number of inputs to a single neuron; 2) separate the data using the separate data program; 3) compute the average of eight chips using the averaging data program; 4) calculate the MSE versus hardness parameter for the collected data; and finally, 5) characterize the sigmoidal transfer function.

G.2 C Program to Collect Data from ETANN

```
/*****************************************************************
* Program: Test to characterize ETANN transfer function using 2
           inputs into one neuron.
*
* Date: 8 July 1992
*
* Author: Capt James Calvin
*****************************************************************/
#include "tsil.h"
#include<stdio.h>
#include<math.h>

main()
{
  REAL8 weight_value[SYNAPSES_PER_NEURON];
  REAL8 bias_value[BIASES_PER_NEURON];
  REAL8 input_value[NEURONS_PER_ARRAY];
  REAL8 etann_output[NEURONS_PER_ARRAY];
  REAL8 actual_bias[ACTUAL_BIASES_PER_NEURON];
  REAL8 etann_weight[SYNAPSES_PER_NEURON];
  REAL8 sigmoid_gain = 3.30;
  REAL8 data = 1.50;
  REAL8 sum[2];

  static REAL8 test_weight[11] = {-2.5, -2.0, -1.5, -1.0, -.5, 0.0,
   .5, 1.0, 1.5, 2.0, 2.5,};
  static REAL8 test1_input[9] = {-1.0, -0.75, -.5, -.25, 0.0, .25,
   .5, .75, 1.0};
```

```
  static REAL8 test2_input[9] = {-1.0, -0.75, -.5, -.25, 0.0, .25,
   .5, .75, 1.0};

  FILE *out_ptr;
  int w,x,y,z;
  StatusT status;

  out_ptr = fopen("test2.out","w");

/****************************************
* Initialize inputs and weights to zero
****************************************/

  for(x = 0; x < NEURONS_PER_ARRAY; x++)
    input_value[x] = 0.0;
  for(x = 0; x < SYNAPSES_PER_NEURON; x++)
    weight_value[x] = 0.0;

/****************************************
*  Connect the ETANN Chip
****************************************/

  printf("\n connecting etann");

  if (connect_etann()!=OK){
     printf("\n error in connecting");
     exit(0);
  }
  else
     printf("\n successful connection");

/*************************************
*      Set Sigmoid Gain
*************************************/

 set_sigmoid_gain(sigmoid_gain);

/*******************************
*     Set Voltage References
*******************************/
```

```
  set_voltage_reference_input(data);
  set_voltage_reference_output(data);

/*******************************
*        Begin Test
*******************************/

   printf("\n Running Test");
   for (w = 0; w < 11; w++){
       weight_value[0] = test_weight[w];
       weight_value[1] = test_weight[w];
       set_weights(0,0,2,weight_value);
       get_weights(0,0,2,&weight_value[0]);
       for (y = 0; y < 9; y++){
   input_value[0] = test1_input[y];
   input_value[1] = test2_input[y];
           write_neuron_inputs(0,2,&input_value[0]);
           read_neuron_outputs(0,1,&etann_output[0]);
   for (y =0; y < 2; y++)
       sum[0] += weight_value[y] * input_value[y];
   fprintf(out_ptr,"%f %f \n",sum[0],etann_output[0]);
       }
       fprintf(out_ptr,"\n");
   }

/*******************************
*  Disconnect ETANN
*******************************/

  printf("\n Disconnect ETANN");
  disconnect_etann();

/*******************************
*    Close Files
*******************************/

  fclose (out_ptr);
  fclose (out1_ptr);
}
```

G.3 C Program to Separate Collected Data

```
/*****************************************************************
*  Program: Separate collected data.
*  Date: 8 July 1992
*  Author: Capt James Calvin
*****************************************************************/

#include <stdio.h>
#include <math.h>

main()
{

FILE *in_ptr;
FILE *out1_ptr, *out2_ptr, *out3_ptr, *out4_ptr, *out5_ptr,
     *out6_ptr, *out7_ptr, *out8_ptr, *out9_ptr, *out10_ptr,
     *out11_ptr;
int x;

  float act_1[10], act_2[10], act_3[10], act_4[10], act_5[10];
  float act_6[10], act_7[10], act_8[10], act_9[10], act_10[10],
        act_11[10];
  float out_1[10], out_2[10], out_3[10], out_4[10], out_5[10];
  float out_6[10], out_7[10], out_8[10], out_9[10], out_10[10],
        out_11[10];

in_ptr = fopen("chiph02.dat","r");
        if (in_ptr==NULL)
      printf("error in file 1");

        out1_ptr = fopen("h1.dat","w");
        if (out1_ptr==NULL)
           printf("error in outfile");

        out2_ptr = fopen("h2.dat","w");
        if (out2_ptr==NULL)
           printf("error in outfile");

        out3_ptr = fopen("h3.dat","w");
        if (out3_ptr==NULL)
           printf("error in outfile");
```

```
    out4_ptr = fopen("h4.dat","w");
    if (out4_ptr==NULL)
       printf("error in outfile");

    out5_ptr = fopen("h5.dat","w");
    if (out5_ptr==NULL)
       printf("error in outfile");

    out6_ptr = fopen("h6.dat","w");
    if (out6_ptr==NULL)
       printf("error in outfile");

    out7_ptr = fopen("h7.dat","w");
    if (out7_ptr==NULL)
       printf("error in outfile");

    out8_ptr = fopen("h8.dat","w");
    if (out8_ptr==NULL)
       printf("error in outfile");

    out9_ptr = fopen("h9.dat","w");
    if (out9_ptr==NULL)
       printf("error in outfile");

    out10_ptr = fopen("h10.dat","w");
    if (out10_ptr==NULL)
       printf("error in outfile");

    out11_ptr = fopen("h11.dat","w");
    if (out11_ptr==NULL)
       printf("error in outfile");

    for (x = 0; x < 9; x++)
fscanf(in_ptr,"%f %f \n", &act_1[x], &out_1[x]);
    fscanf(in_ptr,"\n");
    for (x = 0; x < 9; x++)
fscanf(in_ptr,"%f %f \n", &act_2[x], &out_2[x]);
    fscanf(in_ptr,"\n");
    for (x = 0; x < 9; x++)
fscanf(in_ptr,"%f %f \n", &act_3[x], &out_3[x]);
    fscanf(in_ptr,"\n");
    for (x = 0; x < 9; x++)
```

```
    fscanf(in_ptr,"%f %f \n", &act_4[x], &out_4[x]);
        fscanf(in_ptr,"\n");
        for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_5[x], &out_5[x]);
        fscanf(in_ptr,"\n");
        for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_6[x], &out_6[x]);
        fscanf(in_ptr,"\n");
        for (x = 0; x < 9; x++)
            fscanf(in_ptr,"%f %f \n", &act_7[x], &out_7[x]);
        fscanf(in_ptr,"\n");
        for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_8[x], &out_8[x]);
        fscanf(in_ptr,"\n");
        for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_9[x], &out_9[x]);
        fscanf(in_ptr,"\n");
        for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_10[x], &out_10[x]);
fscanf(in_ptr,"\n");
        for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_11[x], &out_11[x]);
        for (x = 0; x < 9; x++){
    fprintf(out1_ptr,"%f %f \n", act_1[x], out_1[x]);
    fprintf(out2_ptr,"%f %f \n", act_2[x], out_2[x]);
    fprintf(out3_ptr,"%f %f \n", act_3[x], out_3[x]);
    fprintf(out4_ptr,"%f %f \n", act_4[x], out_4[x]);
    fprintf(out5_ptr,"%f %f \n", act_5[x], out_5[x]);
    fprintf(out6_ptr,"%f %f \n", act_6[x], out_6[x]);
    fprintf(out7_ptr,"%f %f \n", act_7[x], out_7[x]);
    fprintf(out8_ptr,"%f %f \n", act_8[x], out_8[x]);
    fprintf(out9_ptr,"%f %f \n", act_9[x], out_9[x]);
    fprintf(out10_ptr,"%f %f \n", act_10[x], out_10[x]);
    fprintf(out11_ptr,"%f %f \n", act_11[x], out_11[x]);
}

  fclose (out1_ptr);
  fclose (out2_ptr);
  fclose (out3_ptr);
  fclose (out4_ptr);
  fclose (out5_ptr);
  fclose (out6_ptr);
```

```
    fclose (out7_ptr);
    fclose (out8_ptr);
    fclose (out9_ptr);
    fclose (out10_ptr);
    fclose (out11_ptr);
    fclose (in_ptr);

}/* end main */
```

G.4 C Program to Calculate Average Output of 8 Chips

```
/*****************************************************************/
/*  Program: Calculate the average output of 8 ETANN chips.
/*  Date: 7 July 1992
/*  Author: Capt James Calvin
/*****************************************************************/

#include <stdio.h>
#include <math.h>

main()
{
  FILE *in1_ptr, *in2_ptr, *in3_ptr, *in4_ptr;
  FILE *in5_ptr, *in6_ptr, *in7_ptr, *in8_ptr, *out1_ptr,
       *out2_ptr;
  int x = 0;

  float act_avg[10], act_1[10], act_2[10], act_3[10];
  float act_4[10], act_5[10], act_6[10], act_7[10], act_8[10];

  float out_avg[10], out_1[10], out_2[10], out_3[10], out_4[10];
  float out_5[10], out_6[10], out_7[10], out_8[10], hardness[10];

  float act1, act2, act3, act4, act5, act6, act7, act8;
  float out1, out2, out3, out4, out5, out6, out7, out8;

  in1_ptr = fopen("a3.dat","r");
  if (in1_ptr==NULL)
    printf("error in file 1");

  in2_ptr = fopen("b3.dat","r");
  if (in2_ptr==NULL)
    printf("error in file 2");

  in3_ptr = fopen("c3.dat","r");
  if (in3_ptr==NULL)
    printf("error in file 3");

  in4_ptr = fopen("d3.dat","r");
  if (in4_ptr==NULL)
    printf("error in file 4");
```

```
in5_ptr = fopen("e3.dat","r");
if (in5_ptr==NULL)
  printf("error in file 5");

in6_ptr = fopen("f3.dat","r");
if (in6_ptr==NULL)
  printf("error in file 6");

in7_ptr = fopen("g3.dat","r");
if (in7_ptr==NULL)
  printf("error in file 7");

in8_ptr = fopen("h3.dat","r");
if (in8_ptr==NULL)
  printf("error in file 8");

out1_ptr = fopen("avg3.dat","w");
if (out1_ptr==NULL)
  printf("error in outfile");

out2_ptr = fopen("hard.dat","a");
if (out2_ptr==NULL)
  printf("error in hardness outfile");


for (x = 0; x < 9;x++){
    fscanf(in1_ptr,"%f %f \n", &act_1[x], &out_1[x]);
    printf("Values from file one: %f %f \n", act_1[x],
                    out_1[x]);
    fscanf(in2_ptr,"%f %f \n", &act_2[x], &out_2[x]);
    printf("Values from file two: %f %f \n", act_2[x],
                    out_2[x]);
    fscanf(in3_ptr,"%f %f \n", &act_3[x], &out_3[x]);
    printf("Values from file three: %f %f \n", act_3[x],
    out_3[x]);
    fscanf(in4_ptr,"%f %f \n", &act_4[x], &out_4[x]);
    printf("Values from file four: %f %f \n", act_4[x],
    out_4[x]);
    fscanf(in5_ptr,"%f %f \n", &act_5[x], &out_5[x]);
    printf("Values from file five: %f %f \n", act_5[x],
    out_5[x]);
    fscanf(in6_ptr,"%f %f", &act_6[x], &out_6[x]);
```

```
      printf("Values from file six: %f %f \n", act_6[x],
      out_6[x]);
      fscanf(in7_ptr,"%f %f", &act_7[x], &out_7[x]);
      printf("Values from file seven: %f %f \n", act_7[x],
      out_7[x]);
      fscanf(in8_ptr,"%f %f", &act_8[x], &out_8[x]);
      printf("Values from file eight: %f %f \n", act_8[x],
      out_8[x]);
   }

  for (x = 0;x < 9;x ++){
     act_avg[x] = (act_1[x] + act_2[x] + act_3[x] + act_4[x] +
            act_5[x] + act_6[x] + act_7[x] + act_8[x]) /
     8.0;
out_avg[x] = (out_1[x] + out_2[x] + out_3[x] + out_4[x] +
            out_5[x] + out_6[x] + out_7[x] + out_8[x]) /
     8.0;
   }

  for (x = 0;x < 9;x ++){
       fprintf(out1_ptr,"%.3f %.3f \n", act_avg[x], out_avg[x]);
       printf("%.3f %.3f \n", act_avg[x], out_avg[x]);
     if (act_avg[x] != 0.0){
  hardness[x] = -(float)log((double)((2/(out_avg[x] + 1))
              - 1)) / act_avg[x];
  fprintf (out2_ptr,"%f \n", hardness[x]);
  printf ("%f \n", hardness[x]);
      }
   }

  fclose (in1_ptr);
  fclose (in2_ptr);
  fclose (in3_ptr);
  fclose (in4_ptr);
  fclose (in5_ptr);
  fclose (in6_ptr);
  fclose (in7_ptr);
  fclose (in8_ptr);
  fclose (out1_ptr);
  fclose (out2_ptr);

}/* end main */
```

G.5 C Program to Calculate MSE vs. Hardness Parameter

```
/*****************************************************************
*  Program: Program to calculate the total MSE vs. hardness.
*  Date: 9 July 1992
*  Author: Capt James Calvin
*****************************************************************/

#include <stdio.h>
#include <math.h>

main()
{
FILE *in_ptr, *out_ptr;
int x, y;

  float act_1[10], act_2[10], act_3[10], act_4[10], act_5[10];
  float act_6[10], act_7[10], act_8[10], act_9[10], act_10[10],
        act_11[10];
  float out_1[10], out_2[10], out_3[10], out_4[10], out_5[10];
  float out_6[10], out_7[10], out_8[10], out_9[10], out_10[10],
        out_11[10];
  float error = 0.0, sum_error = 0.0, sigmoid[90],
        hardness = 0.00;

/*****************************************************************
*                        Open Files
*****************************************************************/

in_ptr = fopen("average.dat","r");
        if (in_ptr==NULL)
      printf("error in file 1");
out_ptr = fopen("mse.dat","w");
   if (out_ptr==NULL)
     printf("error in outfile");

/*****************************************************************
*                        Read in Data
*****************************************************************/

for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_1[x], &out_1[x]);
fscanf(in_ptr,"\n");
```

```
for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_2[x], &out_2[x]);
fscanf(in_ptr,"\n");
for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_3[x], &out_3[x]);
fscanf(in_ptr,"\n");
for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_4[x], &out_4[x]);
fscanf(in_ptr,"\n");
for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_5[x], &out_5[x]);
fscanf(in_ptr,"\n");
for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_6[x], &out_6[x]);
fscanf(in_ptr,"\n");
for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_7[x], &out_7[x]);
fscanf(in_ptr,"\n");
for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_8[x], &out_8[x]);
fscanf(in_ptr,"\n");
for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_9[x], &out_9[x]);
fscanf(in_ptr,"\n");
for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_10[x], &out_10[x]);
fscanf(in_ptr,"\n");
for (x = 0; x < 9; x++)
    fscanf(in_ptr,"%f %f \n", &act_11[x], &out_11[x]);

/*****************************************************************
*             Find Total MSE vs. Hardness Factor
*****************************************************************/

  for (y = 0; y < 90; y++){
      hardness += 0.02;
      sum_error = 0.0;
      for (x = 0; x < 9; x++){
  sigmoid[x] = 1.83/(1 + (float)exp(-(double)hardness*
               act_1[x])) - 0.94;
  error = out_1[x] - sigmoid[x];
  error *= error;
```

```
sum_error += error;
    }
    for (x = 0; x < 9; x++){
sigmoid[x] = 1.83/(1 + (float)exp(-(double)hardness*
             act_2[x])) - 0.94;
error = out_2[x] - sigmoid[x];
error *= error;
sum_error += error;
    }
    for (x = 0; x < 9; x++){
sigmoid[x] = 1.83/(1 + (float)exp(-(double)hardness*
             act_3[x])) - 0.94;
error = out_3[x] - sigmoid[x];
error *= error;
sum_error += error;
    }

    for (x = 0; x < 9; x++){
sigmoid[x] = 1.83/(1 + (float)exp(-(double)hardness*
             act_4[x])) - 0.94;
error = out_4[x] - sigmoid[x];
error *= error;
sum_error += error;
    }
    for (x = 0; x < 9; x++){
sigmoid[x] = 1.83/(1 + (float)exp(-(double)hardness*
             act_5[x])) - 0.94;
error = out_5[x] - sigmoid[x];
error *= error;
sum_error += error;
    }
    for (x = 0; x < 9; x++){
sigmoid[x] = 1.83/(1 + (float)exp(-(double)hardness*
             act_6[x])) - 0.94;
error = out_6[x] - sigmoid[x];
error *= error;
sum_error += error;
    }
    for (x = 0; x < 9; x++){
sigmoid[x] = 1.83/(1 + (float)exp(-(double)hardness*
             act_7[x])) - 0.94;
error = out_7[x] - sigmoid[x];
```

```
    error *= error;
    sum_error += error;
        }
        for (x = 0; x < 9; x++){
    sigmoid[x] = 1.83/(1 + (float)exp(-(double)hardness*
                act_8[x])) - 0.94;
    error = out_8[x] - sigmoid[x];
    error *= error;
    sum_error += error;
        }
        for (x = 0; x < 9; x++){
    sigmoid[x] = 1.83/(1 + (float)exp(-(double)hardness*
                act_9[x])) - 0.94;
    error = out_9[x] - sigmoid[x];
    error *= error;
    sum_error += error;
        }
        for (x = 0; x < 9; x++){
    sigmoid[x] = 1.83/(1+(float)exp(-(double)hardness*
                act_10[x])) - 0.94;
    error = out_10[x] - sigmoid[x];
    error *= error;
    sum_error += error;
        }
        for (x = 0; x < 9; x++){
            sigmoid[x] = 1.83/(1+(float)exp(-(double)hardness*
                act_11[x])) - 0.94;
    error = out_11[x] - sigmoid[x];
    error *= error;
    sum_error += error;
        }
        sum_error /= 99;
        fprintf (out_ptr,"%f %f \n", hardness, sum_error);

    }

fclose (in_ptr);
fclose (out_ptr);

} /* end main */
```

Appendix H. *C Tools for INNTS*

H.1 Introduction

This appendix contains a sampling of C programs used in research for interfacing the Neural Graphics simulator with INNTS software and hardware.

H.2 C Program to Write Neural Graphics' Weights to ETANN

```
/*****************************************************************
* Program:  Write Neural Graphics' Weights to ETANN Chip
* Date: 9 Aug 1992
* Author: Capt James Calvin
*****************************************************************/
#include "tsil.h"
#include<stdio.h>
#include<math.h>

main()
{
  REAL8 weight_value_layer1[16];
  REAL8 weight_value_layer2[20];
  REAL8 bias_value_layer1[1];
  REAL8 bias_value_layer2[1];
  REAL8 sigmoid_gain = 3.30;
  REAL8 data = 1.50;
  FILE *in_ptr,*out_ptr;
  ORD2 w, x, y, z, inputs, outputs, hidden;
  char str[80];
  StatusT status;

 if ((in_ptr = fopen("infile.wts","r")) == NULL){
    printf ("I'm not able to open the input file \n");
    exit(-1);
 }
 printf("Enter number of inputs, a space, & number of
       outputs.\n");
 scanf("%d %d", &inputs, &outputs);
 printf("Enter number of hidden neurons.\n");
 scanf("%d", &hidden);
 fscanf (in_ptr,"%s \n", str);
```

```
/****************************************
*  Connect the ETANN Chip
****************************************/

  printf("\n connecting etann");

  if (connect_etann()!=0 ){
     printf("\n error in connecting");
     exit(0);
  }
  else
     printf("\n successful connection");
     set_etann_gain(NORMAL_ETANN_GAIN);
     set_weight_precision(MAX_PRECISION);

/*************************************
*        Set Sigmoid Gain
*************************************/

  set_sigmoid_gain(sigmoid_gain);
  set_voltage_reference_input(data);
  set_voltage_reference_output(data);

/*****************************************
*   Reading and Writing to Layer One
*****************************************/

  for (w = 0; w < hidden; w++){
      for (x = 0; x < inputs; x++)
            fscanf (in_ptr, "%lf", &weight_value_layer1[x]);
      fscanf (in_ptr, "%lf \n", &bias_value_layer1[0]);
      set_etann_parameters(0,FEEDFORWARD);
      set_weights(w,0,inputs,weight_value_layer1);
      set_bias_weights(w,0,1,bias_value_layer1);
  }

/*****************************************************************
*         Reading and Writing to Layer Two
*****************************************************************/

  fscanf (in_ptr,"\n");
  for (w = 0; w < outputs; w++){
```

```
        for (x = 0; x < hidden; x++)
              fscanf (in_ptr,"%lf", &weight_value_layer2[x]);
        fscanf (in_ptr, "%lf \n", &bias_value_layer2[0]);
        set_etann_parameters(0, FEEDBACK);
        set_weights(w,0,hidden,weight_value_layer2);
        set_bias_weights(w,0,1,bias_value_layer2);
    }
    fclose (in_ptr);

/*******************************
*  Disconnect ETANN
*******************************/

    printf("\n Disconnect ETANN");
    disconnect_etann();
    printf ("\n\n Press any key to continue...");
    getch();
} /* end main */
```

H.3 C Program to Write, Read, and Classify Input Data to ETANN

```
/*****************************************************************
* Program:  Write Inputs to ETANN Chip, Read Outputs, and
*           Classify the Winner
* Date: 4 Aug 1992
* Author: Capt James Calvin
*****************************************************************/
#include "tsil.h"
#include<stdio.h>
#include<math.h>

main()
{
  REAL8 in_data[NEURONS_PER_ARRAY];
  REAL8 out_data[NEURONS_PER_ARRAY];
  REAL8 sigmoid_gain = 3.30;
  REAL8 data = 1.50;
  REAL8 max;
  FILE *in_ptr,*out_ptr;
  ORD2 w, x, y, inputs, outputs, hidden;
  ORD2 count1[60],count2[60];
  float class_winner, actual_class, percentage, junk;
  int test_vectors = 60, total = 0, node,z;

  StatusT status;

 if ((in_ptr = fopen("infile","r")) == NULL){
    printf ("I'm not able to open the input file \n");
    exit(-1);
 }
 if ((out_ptr = fopen("outfile","w")) == NULL){
    printf ("I'm not able to open the input file \n");
    exit(-1);
 }

 printf("Enter number of inputs, a space, & number of
       outputs.\n");
 scanf("%d %d", &inputs, &outputs);
 printf("Enter number of hidden neurons.\n");
 scanf("%d", &hidden);
```

```
/****************************************
*  Connect the ETANN Chip
****************************************/

  printf("\n connecting etann");

  if (connect_etann()!=OK){
     printf("\n error in connecting");
     exit(0);
  }
  else
     printf("\n successful connection");

     set_etann_gain(NORMAL_ETANN_GAIN);

/*************************************
*  Set Sigmoid Gain & Vref out/in
*************************************/

 set_sigmoid_gain(sigmoid_gain);
 set_voltage_reference_input(data);
 set_voltage_reference_output(data);

/****************************************
*  Begin Writing Inputs to ETANN
****************************************/

 printf("\n Writing inputs");

/****************************************
*  Initialize the Count
****************************************/

for (x = 1; x <= outputs; x++){
    count1[x] = 0;
    count2[x] = 0;
}

/****************************************
```

```
*   Writing Inputs
*****************************************/

 set_etann_parameters(0,FEEDFORWARD);
 for (x = 0; x < test_vectors; x++){
     fscanf (in_ptr,"%f", &junk);
     for (y = 0; y < inputs; y++)
           fscanf (in_ptr,"%lf", &in_data[y]);
     fscanf (in_ptr,"%f \n", &actual_class);
     write_neuron_inputs(0,inputs,in_data);
     clock_feedback(1);
     read_neuron_outputs(0,outputs,&out_data[0]);
     for (z = outputs - 1; z >= 0; z--){
           w = outputs - z;
           fprintf (out_ptr," Neuron Output for class %d = %.3lf
                    ",w, out_data[z]);
           if ((w % 2) == 0) fprintf (out_ptr,"\n");
     }

/*********************************************************************
*                  Classify the Output
*********************************************************************/
/*Neural Graphics calls neuron 0 the last class (reverse logic)*/

     max = -1.0;
     for (y = 0; y < outputs; y++){
           if (out_data[y] >= max){
              max = out_data[y];
              node = y;
           }
     }
     class_winner =  (float)(outputs - node);
     fprintf (out_ptr,"\n Desired Output = %.0f  Winner = %.0f
              \n\n", actual_class, class_winner);
     if (actual_class == class_winner){
          y = (int)class_winner;
          count1[y] += 1;
          total++;
     }
     for (y = 1; y <= outputs; y++)
           if ((int)actual_class == y) count2[y] += 1;
 }
```

```
/*********************************************************************
*                        Print the Output
*********************************************************************/

 fprintf (out_ptr,"Total Number of Test Vectors = %d \n",
         test_vectors);
 for (x = 1; x <= outputs; x++){
     percentage = ((float)count1[x] / (float)count2[x])*100;
     fprintf (out_ptr,"Class %d: Total = %d  Correct = %d
             Percent = %.2f \n",x,count2[x],count1[x],percentage);
 }
 fprintf (out_ptr,"Total Number of Classified Right = %d \n\n",
         total);
 percentage = ((float)total / (float)test_vectors)*100;
 printf ("\n\n Total Test Vectors = %d", test_vectors);
 printf ("\n\n Total Classified Right = %d", total);
 fprintf (out_ptr,"Percentage Right = %.2f \n", percentage);
 printf ("\n\n Percentage Right = %.2f ", percentage);

/*********************************
*     Disconnect ETANN
*********************************/

  printf("\n Disconnect ETANN");
  disconnect_etann();
  fclose (in_ptr);
  fclose (out_ptr);
  printf ("\n\n Press any key to continue...");
  getch();
} /* end main */
```

H.4 C Program to Write Neural Graphics' Weights to EMB

```
/*****************************************************************
* Program:  Write Neural Graphics' Weights to EMB
* Date: 27 Aug 1992
* Author: Capt James Calvin
*****************************************************************/
#include "tsil.h"
#include<stdio.h>
#include<math.h>

main()
{
  REAL8 weight_value_layer1[16];
  REAL8 weight_value_layer2[35];
  REAL8 bias_value_layer1[6];
  REAL8 bias_value_layer2[6];
  REAL8 bias;
  FILE *in_ptr,*out_ptr;
  ORD2 w, x, y, z, inputs, outputs, hidden;
  char str[80];
  StatusT status;

 if ((in_ptr = fopen("infile.wts","r")) == NULL){
    printf ("I'm not able to open the input file \n");
    exit(-1);
 }

 printf("Enter number of inputs, a space, & number
       of outputs.\n");
 scanf("%d %d", &inputs, &outputs);
 printf("Enter number of hidden neurons.\n");
 scanf("%d", &hidden);
 fscanf (in_ptr,"%s \n", str);


/****************************************
*       Connect the EMB
****************************************/

  printf("\n Place your EMB in the adapter socket, then press
         a key...");
  getch();
```

```
  if (connect_emb(i80170_NX) != OK){
     printf("\n error in connecting");
     exit(0);
  }
  else
     printf("\n successful connection");
     set_etann_gain(NORMAL_ETANN_GAIN);
     set_weight_precision(MAX_PRECISION);

/****************************************
*   Reading and Writing to Layer One
****************************************/

  for (w = 0; w < hidden; w++){
      for (x = 0; x < inputs; x++)
  fscanf (in_ptr, "%lf", &weight_value_layer1[x]);
      fscanf (in_ptr, "%lf \n", &bias);
      for (x = 0; x < 6; x++)
  bias_value_layer1[x] = bias / 6.0;
      set_etann_parameters(0,FEEDFORWARD);
      set_weights(w,0,inputs,weight_value_layer1);
      set_bias_weights(w,0,6,bias_value_layer1);
  }

/*****************************************************************
* Reading and Writing to Layer Two
*****************************************************************/

  fscanf (in_ptr,"\n");
  for (w = 0; w < outputs; w++){
      for (x = 0; x < hidden; x++)
  fscanf (in_ptr,"%lf", &weight_value_layer2[x]);
      fscanf (in_ptr, "%lf \n", &bias);
      for (x = 0; x < 6; x++)
  bias_value_layer2[x] = bias / 6.0;
      set_etann_parameters(0, FEEDBACK);
      set_weights(w,0,hidden,weight_value_layer2);
      set_bias_weights(w,0,6,bias_value_layer2);
  }
  fclose (in_ptr);
```

```
/*********************************
*  Disconnect ETANN
*********************************/

  printf("\n Disconnect EMB");
  disconnect_emb();
  printf ("\n\n Press any key to continue...");
  getch();
} /* end main */
```

H.5 C Program to Write, Read, and Classify Input Data to EMB

```
/*****************************************************************
* Program:  Write Inputs to EMB and Read Outputs
* Date: 27 Aug 1992
* Author: Capt James Calvin
*****************************************************************/
#include "tsil.h"
#include<stdio.h>
#include<math.h>

main()
{
  REAL8 in_data[16];
  REAL8 out_data[30];
  REAL8 max;
  FILE *in_ptr,*out_ptr;
  ORD2 w, x, y, inputs, outputs, hidden;
  ORD2 count1[31],count2[31];
  float class_winner, actual_class, percentage, junk;
  int test_vectors = 750, total = 0, node,z;

  StatusT status;

 if ((in_ptr = fopen("infile","r")) == NULL){
    printf ("I'm not able to open the input file \n");
    exit(-1);
 }
 if ((out_ptr = fopen("outfile","w")) == NULL){
    printf ("I'm not able to open the input file \n");
    exit(-1);
 }

 printf("Enter number of inputs, a space, & number of
         outputs.\n");
 scanf("%d %d", &inputs, &outputs);

/***************************************
*        Connect the EMB
***************************************/

  printf("\n Place your EMB in the adapter socket, then press
```

```
        a key...");
  getch();
  if (connect_emb(i80170_NX) != OK){
     printf("\n error in connecting");
     exit(0);
  }
  else
     printf("\n successful connection");

     set_etann_gain(NORMAL_ETANN_GAIN);

/****************************************
*  Initialize the Count
****************************************/

for (x = 1; x <= outputs; x++){
    count1[x] = 0;
    count2[x] = 0;
}


/****************************************/

/****************************************
*   Writing Inputs
****************************************/
 printf("\n Writing inputs");
 set_etann_parameters(0,FEEDFORWARD);
 for (x = 0; x < test_vectors; x++){
     fscanf (in_ptr,"%f", &junk);
     for (y = 0; y < inputs; y++)
 fscanf (in_ptr,"%lf", &in_data[y]);
     fscanf (in_ptr,"%f \n", &actual_class);
     write_neuron_inputs(0,inputs,in_data);
     clock_feedback(1);
     read_neuron_outputs(0,outputs,&out_data[0]);
     for (z = outputs - 1; z >= 0; z--){
 w = outputs - z;
 fprintf (out_ptr," Neuron Output for class %d =
        %.3lf ",w, out_data[z]);
 if ((w % 2) == 0) fprintf (out_ptr,"\n");
     }
```

```
/*****************************************************************
*                  Classify the Output
*****************************************************************/
/* Neural Graphics calls neuron 0 highest class (reverse order)*/

     max = -1.0;
     for (y = 0; y < outputs; y++){
 if (out_data[y] >= max){
    max = out_data[y];
    node = y;
 }
     }
     class_winner =  (float)(outputs - node);
     fprintf (out_ptr,"\n Desired Output = %.0f  Winner =
             %.0f \n\n",
     actual_class, class_winner);
     if (actual_class == class_winner){
y = (int)class_winner;
count1[y] += 1;
total++;
     }
     for (y = 1; y <= outputs; y++)
 if ((int)actual_class == y) count2[y] += 1;
 }
/*****************************************************************
*                       Print the Output
*****************************************************************/

 fprintf (out_ptr,"Total Number of Test Vectors = %d \n",
         test_vectors);
 for (x = 1; x <= outputs; x++){
     percentage = ((float)count1[x] / (float)count2[x])*100;
     fprintf (out_ptr,"Class %d: Total = %d  Correct = %d
             Percent = %.2f \n",x,count2[x],count1[x],percentage);
 }
 fprintf (out_ptr,"Total Number of Classified Right = %d
         \n\n", total);
 percentage = ((float)total / (float)test_vectors)*100;
 printf ("\n\n Total Test Vectors = %d", test_vectors);
 printf ("\n\n Total Classified Right = %d", total);
 fprintf (out_ptr,"Percentage Right = %.2f \n", percentage);
 printf ("\n\n Percentage Right = %.2f ", percentage);
```

```
/*******************************
*      Disconnect EMB
*******************************/

  printf("\n Disconnect EMB");
  disconnect_emb();
  fclose (in_ptr);
  fclose (out_ptr);
  printf ("\n\n Press any key to continue...");
  getch();
}
```

Bibliography

1. Ardizzone, E. and others. "A Digital Architecture Implementing the Self-Organizing Feature Maps," *Artificial Neural Networks.* Elsevier Science Publishers B.V. (North-Holland), 1991.
2. Burckel, William M. *Modularly Expandable Artificial Neural Network (MEANN) Computer for Development and Application of Large Scale ANNs.* DTIC Report: AD-B149740L (October 1990).
3. California Scientific Software. *BrainMaker Professional* (2nd Edition). Grass Valley, CA (December 1990).
4. Castro, H.A., S.M. Tam, and M. Holler. "Implementation and Performance of an Analog Non-Volatile Neural Network," *80170NX Neural Network Technology and Applications.* Intel Corporation, Santa Clara, CA, 1992.
5. Cameron, Capt (CAF) David M. *Radar System Classification using Neural Networks.* MS Thesis AFIT/GSO/ENS/91D-03. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH (December 1991).
6. Chan, L.W. and F. Fallside. "An Adaptive Training Algorithm for Back Propagation Networks," *Computer Speech and Language.* 2: 205–218 (1987).
7. Eppler, Wolfgang and others. "A Digital Signal Processor for Simulating Back-Propagation Networks," *Artificial Neural Networks.* Elsevier Science Publishers B.V. (North-Holland), 1991.
8. Graf, H.P. and others. "A Neural-Net Board System for Machine Vision Applications," *International Joint Conference on Neural Networks* (Sponsored by IEEE and INNS). I:481-486 (July 1991).
9. Halonen, Kari and others. "VLSI Implementation of a Programmable Dual Computing Cellular Neural Network Processor," *Artificial Neural Networks.* Elsevier Science Publishers B.V. (North-Holland),1991.
10. Harrer, Hubert. "An Analog Implementation of Discrete-Time Cellular Neural Networks," *IEEE Transactions on Neural Networks.* 3:466-475.
11. Howitt, Ivan. "Radar Warning Receiver Emitter Identification Processing Utilizing Artificial Neural Networks," *SPIE Vol. 1294 Applications of Artificial Neural Networks* (1991).
12. Heemskerk, Jan N.H. and others. "The BSP400: A Modular Neuralcomputer Assembled from 400 Low-Cost Microprocessors," *Artificial Neural Networks.* Elsevier Science Publishers B.V.(North-Holland), 1991.
13. Intel Corporation. *80170NX Electronically Trainable Analog Neural Network.* Product Manual, Order Number: 290408-002, Intel Corporation, Santa Clara, CA (June 1991).

14. Jackson, Darin and Dan Hammerstrom. "Distributing Back Propagation Networks Over the Intel i-PSC/860 Hypercube," *International Joint Conference on Neural Networks* (Sponsored by IEEE and INNS). I: 569-574 (July 1991).

15. Klimasauskas, Casimir C. "Neural Networks: A Short Course," *PC AI* Magazine. 6:26-30 (November/December 1988).

16. Kohonen, Teuvo. "The SelfOrganizing Map," *Proceedings of the IEEE*, *78*:1464-1480 (September 1990).

17. Lippmann, Richard P. "An Introduction to Computing with Neural Nets," *IEEE ASSP* Magazine (April 1987).

18. Lont, Jerzy B. and Walter Guggenbuhl. "Analog CMOS Implementation of a Multilayer Perceptron with Nonlinear Synapses," *IEEE Transactions on Neural Networks.* 3:457-464 (May 1992).

19. Mann, Richard and Simon Haykin. "Application of the Self-Organising Feature Map and Learning Vector Quantisation to Radar Clutter Classification," *Artificial Neural Networks*, Elsevier Science Publishers B.V. (North-Holland), 1991.

20. Martin, Fin. Personal Correspondence. Intel Corporation, Santa Clara, CA, 26 March 1992.

21. Melton, Matthew S. and others. "The TIn-MANN VLSI Chip." *IEEE Transactions on Neural Networks.* 3:375-383 (May 1992).

22. Mumford, Michael L., David K. Andes, and Lynn R. Kern. "The Mod 2 Neurocomputer System Design," *IEEE Transactions on Neural Networks.* 3:423-433 (May 1992).

23. NeuroDynamX, Inc. *iDynaMind* (Version 2.0). User's Guide, South Pasadena, CA, 1991.

24. O'Neill, Bryan. "Using Neural Networks in Receiver Processing," *Defense Computing*, (January–February 1989).

25. Palovpuori, Karri and others. "VLSI Implementation of a Pulse-Density Modulated Neural Network for PC-Controlled Computing Environment," *Artificial Neural Networks.* Elsevier Science Publishers B.V. (North-Holland), 1991.

26. Raeth, Peter G. "Raising the Question of Backpropagation's Logistical Supportability," Unpublished Paper, 1991.

27. Rogers, Steven K. and others. *An Introduction to Biological and Artificial Neural Networks.* Wright-Patterson AFB, OH: Air Force Institute of Technology (23 October 1990).

28. Rogers, Steven K. and Matthew Kabrisky. *An Introduction to Biological Artificial Neural Networks for Pattern Recognition.*TT4, SPIE Optical Engineering Press, 1991.

29. Rosenblatt. *Principles of Neurodynamics.* New York: Spartan Books, 1959.

30. Ruck, Capt Dennis W. *Characterization of Multilayer Perceptrons and their Application to Multisensor Automatic Target Detection.* Doctoral Dissertation. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1990 (AD-A229 035).

31. Sackinger, Eduard and others. "Application of the ANNA Neural Network Chip to High-Speed Character Recognition," *IEEE Transactions on Neural Networks.* 3:498-504 (May 1992).

32. Sankar, Ananth and Richard J. Mammone. "Optimal Pruning of Neural Tree Networks for Improved Generalization," *International Joint Conference on Neural Networks.* (Sponsored by IEE and INNS), II: 219–223 (July 1991).

33. Sato, Akira. "An Analytical Study of the Momentum Term in a Backpropagation Algorithm," *Artificial Neural Networks.* Elsevier Science Publishers B.V. (North-Holland), 1991.

34. Stanley, Jeanette. *Introduction to Neural Networks* (Third Edition). Sierra Madre: California Scientific Software, 1990.

35. Siggelkow, Andreas and others. "Influence of Hardware Characteristics on the Performance of a Neural System, *Artificial Neural Networks.* Elsevier Science Publishers B.V. (North-Holland),1991.

36. Tam, Simon and others. "A Reconfigurable Multi-Chip Analog Neural Network: Recognition and Back-Propagation Training." *IJCNN.* II:625–630 (June 1992).

37. Tarr, Gregory L. and others. "AFIT Neural Network Development Tools and Techniques for Modeling Artificial Neural Networks," *Proceedings of SPIE.* 1294:211–216. Bellingham, WA: SPIE—The International Society for Optical Engineering, 1990.

38. Tarr, Gregory L. and others. "Effective Neural Network Modeling in C," *Artificial Neural Networks.* Elsevier Science Publishers B.V.(North-Holland), 1991.

39. Tomberg, Jouni and Kimmo Kaski. "Digital VLSI Architecture of Backpropagation Algorithm with On-Chip Learning," *Artificial Neural Networks.* Elsevier Science Publishers B.V. (North-Holland), 1991.

40. Wasserman, Philip D. *Neural Computing: Theory and Practice.* New York: Van Nostrand Reinhold, 1989.

41. Widrow, Bernard and Michael Lehr. "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation," *Proceedings of the IEEE.* 78: 1415–1442 (September 1990).

42. Willson, Gregory B. "Radar Classification Using a Neural Network," *SPIE Vol. 1294 Applications of Artificial Neural Networks* (1990).

43. Zahirniak, Daniel R. *Characterization of Radar Signals Using Neural Networks.* Master's Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1990 (AD-A230 582).

Vita

Captain James B. Calvin, Jr., was born on 20 December 1958 in Jackson, California. After graduating from Highland High School in Bakersfield, California, he enlisted in the United States Air Force, 31 August 1977, and entered active duty on 3 October 1977. Until June 1984, he served as an aircraft electrician and quality assurance inspector/evaluator, attaining the rank of technical sergeant. James was selected in March 1984 for the Airman's Education and Commissioning Program (AECP) and in June 1984 entered the Georgia Institute of Technology where he earned the degree of Bachelor of Electrical Engineering, December 1986. Following graduation, he attended USAF Officer Training School where he was commissioned as a second lieutenant. He was subsequently assigned to the Foreign Technology Division (FTD), Wright-Patterson AFB, Ohio. James served over four years at FTD as a Space Electronics Engineer working in the area of foreign electronic space communications. Also during this same time period, he earned a Master of Science in Administration from Central Michigan University. In June 1991, James entered the School of Engineering to pursue a master's degree in electrical engineering at the Air Force Institute of Technology (AU), Wright-Patterson AFB, Ohio. He has a follow-on assignment to Robins AFB, Georgia, as an Electronic Warfare Systems Engineer.

Permanent address: P.O. Box 82
Wilseyville, California
95257

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE
ETANN HARDWARE IMPLEMENTATION FOR RADAR EMITTER IDENTIFICATION

5. FUNDING NUMBERS

6. AUTHOR(S)
Captain James B. Calvin, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/92D-08

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Captain Daniel Zahirniak
WDRC/AAWP-1
WPAFB, OH 45433-6543

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This study investigated classification of 30 radar emitters with 16 signal features using Intel's 80170NX chip, the Electronically Trainable Analog Neural Network (ETANN). Software tools were developed to characterize the ETANN sigmoidal transfer function for use in a custom simulator, known as Neural Graphics. Neural Graphics operates on a Silicon Graphics workstation. The Intel Neural Network Training System simulators were used in early experiments, but were found to be inefficient in training on data used in this research. Using a modified Neural Graphics simulator, single chip and multi-chip experiments were performed to provide benchmark results prior to performing chip-in-loop training. By maximizing off-chip training accuracy, the need for on-chip training is minimized and therefore the device life is prolonged. Several single chip and multi-chip configurations were tried; the final architecture which produced the maximum on-chip classification accuracy was a hierarchical network. The maximum on-chip classification accuracy for a single chip implementation of 30 classes without chip-in-loop training was 83 percent. Again without chip-in-loop training, the maximum on-chip classification accuracy for a hierarchical configuration with the 30-class problem was 87 percent.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Radar Emitter Identification, ETANN, Neural Network Hardware	155
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL